Microsoft® QuickBASIC 4

BASIC-Befehlsverzeichnis

Microsoft®

Microsoft® QuickBASIC

BASIC-Befehlsverzeichnis

Für IBM® Personal Computer
und kompatible Computer

Microsoft Corporation

Artikelnr. 01646
Dokumentnr. 8623-400-00-1

Inhaltsverzeichnis

Teil 3: Anhänge



Einleitung

Microsoft hat BASIC zu einer umfassenden Programmiersprache entwickelt, die sich sowohl für kommerzielle Anwendungen als auch für schnelle, individuelle Programme eignet. Die erweiterten Eigenschaften von QuickBASIC unterstützen benutzerdefinierte Datentypen, Rekursion und weitere Eigenschaften, die es zu einer attraktiven Sprache für professionelle Anwender machen. Gleichzeitig werden durch die Kompatibilität zu BASICA die Teile der Sprache erhalten, die BASIC zu einer idealen Sprache für all diejenigen gemacht haben, die mit der Programmierung beginnen.

QuickBASIC bietet die folgenden neuen Eigenschaften:

- **Unterprogramme und Funktionen**

  Unterprogrammme erlauben es Ihnen, Programme in unterschiedliche Module aufzuteilen. Sie können Unterprogramme verwenden, um häufig gestellte Aufgaben beliebig oft in verschiedenen Programmen zu erfüllen. Die **FUNCTION**-Prozeduren von QuickBASIC bieten komplette, flexible, benutzerdefinierte Funktionen, die eine gute Alternative zu den **DEF FN**-Anweisungen bieten.

- **Benutzerdefinierte Datentypen**

  Benutzerdefinierte Datentypen erlauben eine Kombination von numerischen Variablen und Zeichenkettenvariablen zu neuen zusammengesetzten Variablentypen. Sie können diese neuen Typen zur Vereinfachung von Direktzugriffsdatei-E/A und zum Erstellen von komplexeren Datenstrukturen verwenden.

- **Rekursion**

  QuickBASIC erlaubt das Schreiben von rekursiven Prozeduren - Prozeduren, die sich selbst aufrufen können. Damit sind Sie in der Lage, die vielen wichtigen rekursiven Algorithmen (z. B. Quicksort) in Ihren BASIC-Programmen zu verwenden.

- **Flexible Datenfelddimensionierung**

  QuickBASIC-Datenfelder können positive und negative Indizes haben. Die Datenfelder müssen nicht länger mit 0 oder 1 beginnen - sie können mit dem Index beginnen, der am besten zu dem von Ihnen geschriebenen Programm paßt.

- **Binärdateien**

  QuickBASIC unterstützt Binärdatei-E/A. Somit können Ihre BASIC-Programme jetzt Dateien in jedem Format erstellen und manipulieren, von Textverarbeitungsdateien bis zu Bibliotheken, da Binärdatei-E/A es Ihrem Programm erlaubt, jede Anzahl von Bytes an beliebiger Stelle in eine Datei zu schreiben oder aus ihr zu lesen.

Überblick über dieses Handbuch

Das Microsoft® BASIC-Befehlsverzeichnis beschreibt die Programmiersprache QuickBASIC. Benutzen Sie es während der Programmierung als Referenz, um Informationen über Syntax und Auswirkung einer Anweisung zu finden. Sie können es ebenso verwenden, um BASIC zu erforschen, indem Sie Ihnen unvertraute Anweisungen durchlesen und in einem kleinen Programm ausprobieren.

Dieses Buch versucht nicht, Sie in BASIC, Programmieren oder in der Benutzung der QuickBASIC-Umgebung zu unterrichten. Mehr über BASIC und Programmierung finden Sie in *Programmieren in BASIC: Ausgewählte Themen* und in den Büchern unter "Einleitung". Informationen zur Anwendung der vielen Möglichkeiten der QuickBASIC-Umgebung finden Sie in *Lernen und Anwenden von Microsoft QuickBASIC*.

Das Sprachhandbuch besteht aus drei Teilen:

- Teil 1, "Sprachgrundlagen", beschreibt die Elemente der Sprache BASIC: Zeichen, Programmzeilen, Datentypen, Konstanten, Variablen, Module, Programme und Prozeduren.
- Teil 2, "Nachschlageteil - Anweisungen und Funktionen", enthält eine Sammlung aller BASIC-Anweisungen und -Funktionen.
- Teil 3, "Anhänge", enthält weitere Anhänge, die die Tastaturabfrage-Codes und die ASCII-Zeichen, die reservierten BASIC-Wörter, Metabefehle und Fehlermeldungen beschreiben.

Typographische Konventionen

In diesem Handbuch werden folgende typographische Konventionen verwendet:

Darstellung	*Beschreibung*
`Beispiele` **`Eingabe:`**	Die in der linken Spalte gezeigte Schriftart wird zur Darstellung von Informationen benutzt, die auf Ihrem Bildschirm erscheinen. Die fettgedruckte Version dieses Schrifttyps zeigt an, daß eine Eingabe nach einem Eingabeaufforderungszeichen eingegeben wird.
Apostroph (')	Das Apostroph kennzeichnet einen Kommentar in Beispielprogrammen. Ein Apostroph wird durch Drücken der Taste eingegeben, die ein einfaches rechtes Anführungszeichen erzeugt. Die Taste kann als ' oder " beschriftet sein. Benutzen Sie nicht das einfache linke Anführungszeichen, welches durch die mit ' gekennzeichnete Taste erzeugt wird.
SCHLÜSSELWÖRTER	Fettgedruckte Großbuchstaben zeigen BASIC-Schlüsselwörter an. Diese Schlüsselwörter sind ein nötiger Bestandteil der Anweisungs-Syntax, es sei denn, sie sind, wie nachfolgend beschrieben, in eckige Klammern eingeschlossen. Wenn Sie Programme schreiben, müssen Sie Schlüsselwörter genau wie gezeigt eingeben. Sie können dabei Groß- und/oder Kleinbuchstaben benutzen.
Befehlsnamen	Fettgedruckte Kleinbuchstaben im Text zeigen Menü- und Befehlsnamen an.
Platzhalter	Die Begriffe in Kursivschrift sind Platzhalter für Informationen, die Sie angeben müssen, wie z. B. einen Dateinamen. Darüber hinaus wird Kursivschrift gelegentlich im Text zur Hervorhebung benutzt.
[**Wahlweise Begriffe**]	Zwischen eckigen Klammern stehende Begriffe können wahlweise verwendet werden.

Darstellung	***Beschreibung***
{*Wahl1* | *Wahl2*}	Geschweifte Klammern und ein vertikaler Balken zeigen an, daß Sie die Wahl zwischen zwei oder mehr Begriffen haben. Sie müssen einen der Begriffe wählen, es sei denn, alle Begriffe sind auch in eckige Klammern eingeschlossen.
Wiederholende Elemente...	Drei Punkte, die einem Begriff folgen, zeigen an, daß mehrere Begriffe derselben Form auftreten können.

`Programm`
.
.
.
`Fragment`
	Eine senkrechte Punktreihe zeigt an, daß ein Teil des Programms weggelassen wurde.

TASTENBEZEICHNUNGEN	Kleingedruckte Großbuchstaben werden für die Bezeichnungen von Tasten und Tastenkombinationen verwendet, wie z.B. EINGABETASTE und STRG+R. Beachten Sie bitte, daß ein Plus (+) eine Kombination von Tasten anzeigt. Zum Beispiel zeigt Ihnen STRG+E, daß die STRG-TASTE gedrückt bleiben muß, während die E-Taste betätigt wird. Die in diesem Handbuch verwendeten Tastenbezeichnungen entsprechen den Bezeichnungen, die auf den Tasten des IBM®-Personal Computers stehen. Falls Sie einen anderen Computer benutzen, können diese Tasten abweichende Bezeichnungen tragen. Die Gruppe der Cursor-Bewegungstasten (auch "Pfeiltasten" genannt), die sich im numerischen Tastenfeld rechts neben der Haupttastatur befinden, werden RICHTUNGSTASTEN genannt. Jede einzelne RICHTUNGSTASTE wird entweder durch die Richtung des Pfeiles auf der Taste bezeichnet (NACH LINKS, NACH RECHTS, NACH OBEN, NACH UNTEN) oder durch die Bezeichnung auf der Taste (BILD ↑, BILD ↓). Die Wagenrücklauftaste wird als EINGABETASTE bezeichnet.
"Definierter Ausdruck"	Anführungsstriche kennzeichnen entweder einen im Text definierten Ausdruck oder den Namen eines Begriffs in einem Dialogfeld.

Hinweis Die folgende Syntax (für die **PRINT**-Anweisung) benutzt viele der typographischen Konventionen in diesem Handbuch.

PRINT [*Ausdrucksliste*] [{, | ; }]

Aufbau der Seiten des Nachschlageteils

Jede Funktions- und Anweisungsbeschreibung in Teil 2, "Nachschlageteil - Anweisungen und Funktionen", benutzt das folgende Format:

Objekt	*Beschreibung*
Funktion	Faßt kurz zusammen, was die Anweisung oder Funktion bewirkt.
Syntax	Gibt die richtige Syntax für die Anweisung oder Funktion.
Anmerkungen	Beschreibt die Argumente und Optionen im einzelnen und erläutert, wie die Anweisung oder Funktion verwendet wird.
Unterschiede zu BASICA	Teilt mit, ob die vorliegende Anweisung oder Funktion neu hinzugekommen ist, erweitert wurde, oder ob sie sich von derselben Anweisung des im IBM Personal Computer-Handbuch beschriebenen Microsoft BASIC 2.0-Interpreters unterscheidet. (Dieser Eintrag ist nicht bei jedem Stichwort zu finden.)
Vergleichen Sie auch	Verweist auf verwandte Anweisungen und Funktionen. (Dieser Eintrag ist nicht bei jedem Stichwort zu finden.)
Beispiel	Gibt Beispiele für Befehle, Programme und Programmteile, die den Gebrauch der Anweisung oder Funktion veranschaulichen. (Dieser Eintrag ist nicht bei jedem Stichwort zu finden.)



Teil 1: Sprachgrundlagen

Teil 1 stellt die Elemente von BASIC vor. Die Kapitel 1-3 beschreiben den BASIC-Zeichensatz, die BASIC-Programmzeile, Datentypen, Konstanten, Variablen, Operatoren und Ausdrücke. Kapitel 4 beschreibt die BASIC-Programmodule und -Prozeduren. Dieses Kapitel erläutert die Beziehungen zwischen Programmen und Modulen und diskutiert Prozedur- und Rekursionstypen.

1 Sprachelemente

Zeichen aus dem BASIC-Zeichensatz werden zusammengestellt zu Marken, Schlüsselwörtern, Variablen und Operatoren. Diese Elemente ihrerseits bilden die Anweisungen, die ein Programm ausmachen.

Dieses Kapitel beschreibt den Zeichensatz und das Format von BASIC-Programmzeilen. Im besonderen behandelt es:

- besondere Zeichen aus dem Zeichensatz und die besonderen Bedeutungen, die manche Zeichen haben
- das Format einer Zeile in einem BASIC-Programm
- Zeilenmarken
- ausführbare und nicht ausführbare Anweisungen
- Programmzeilenlängen

Kapitel 2, "Datentypen", und Kapitel 3, "Ausdrücke und Operatoren", behandeln andere Teile von Anweisungen, einschließlich Ausdrücke.

1.1 Zeichensatz

Der Microsoft QuickBASIC-Zeichensatz besteht aus alphabetischen Zeichen, numerischen Zeichen und Sonderzeichen.

Die alphabetischen Zeichen in BASIC sind die Großbuchstaben (A - Z) und die Kleinbuchstaben (a - z) des Alphabets.

Die numerischen Zeichen in BASIC sind die Ziffern 0 - 9. Die Buchstaben A - F und a - f können als Teile von Hexadezimalzahlen benutzt werden. Die folgenden Zeichen haben besondere Bedeutungen in BASIC-Anweisungen und -Ausdrücken:

Zeichen	***Name oder Funktion***
EINGABETASTE	Beendet die Eingabe einer Zeile
	Leerzeichen (oder Leerschritt)
!	Ausrufezeichen (Anhang für Datentyp einfacher Genauigkeit)
#	Nummern- (oder Pfund-) zeichen (Anhang für Datentyp doppelter Genauigkeit)
$	Dollarzeichen (Anhang für Zeichenketten-Datentyp)
%	Prozentzeichen (Anhang für ganzzahligen Datentyp)
&	Kaufmännisches Und-Zeichen (Anhang für 4-Byte-Ganzzahl-Datentypen)
'	Einfaches Anführungszeichen (Apostroph)

Zeichen	***Name oder Funktion***
(	Runde Klammer auf
)	Runde Klammer zu
*	Stern (Multiplikationszeichen)
+	Pluszeichen
,	Komma
-	Minuszeichen
.	Punkt oder Dezimalpunkt
/	Schrägstrich oder Divisionszeichen
:	Doppelpunkt
;	Semikolon
<	Kleiner als
=	Gleichheitszeichen (Zuweisungszeichen oder Vergleichsoperator)
>	Größer als
?	Fragezeichen
@	At-Symbol
[	Eckige Klammer auf
]	Eckige Klammer zu
\	Rückwärtiger Schrägstrich (Ganzzahl-Divisionszeichen)
^	Aufwärts gerichteter Pfeil (Exponentialzeichen)
_	Unterstreichungszeichen (Zeilenfortsetzung)

1.2 Die BASIC-Programmzeile

BASIC-Programmzeilen haben folgende Syntax:

[*Zeilenkennzeichen*] [*Anweisung*] [: *Anweisung*]...[*Kommentar*]

1.2.1 Wie Sie Zeilenkennzeichen benutzen

BASIC unterstützt zwei Typen von *Zeilenkennzeichen*: Zeilennummern und alphanumerische Zeilenmarken:

1. Eine Zeilennummer kann jede Kombination aus Ziffern von 0 bis 65.529 sein. Die folgenden Nummern sind gültige Zeilennummern:

```
1
200
300 PRINT "HALLO"    '300 ist die Zeilennummer
65000
```

Wichtig Von der Verwendung der Zeilennummer 0 wird abgeraten. Fehler- und Ereignisverfolgungs-Anweisungen (**ON ERROR** und **ON** *Ereignis*) interpretieren eine vorhandene Zeilennummer 0 so, als wäre die Verfolgung ausgeschaltet (gestoppt). Beispielsweise schaltet die nachstehende Anweisung die Fehlerverfolgung aus, verzweigt aber nicht zur Zeile 0, wenn ein Fehler auftritt.

```
ON ERROR GOTO 0
```

Darüber hinaus wird bei **RESUME 0** die Ausführung nicht in Zeile 0, sondern in der Zeile fortgesetzt, in der der Fehler aufgetreten ist.

2. Eine alphanumerische Zeilenmarke kann jede beliebige Kombination aus 1 bis 40 Buchstaben und Ziffern sein, die mit einem Buchstaben anfängt und mit einem Doppelpunkt endet. BASIC-Schlüsselwörter sind nicht zulässig. Dies sind gültige alphanumerische Zeilenmarken:

```
alpha:
ScreenSub:
Test3A:
```

Die Schreibweise (Groß- und Kleinbuchstaben) ist unbedeutend. Folgende Zeilenmarken sind gleichwertig:

```
alpha:
Alpha:
ALPHA:
```

Zeilennummern und Zeilenmarken können in jeder Spalte beginnen, solange sie die ersten Zeichen, außer Leerzeichen oder Tabulatorzeichen, in einer Zeile sind. Leerzeichen und Tabulatorzeichen sind zwischen einer alphanumerischen Marke und dem nachfolgenden Doppelpunkt erlaubt. Eine Zeile kann nur eine Marke haben.

In BASIC muß nicht jede Zeile eines Quellprogramms mit einer Zeilennummer oder einer Zeilenmarke beginnen. Sie können sowohl alphanumerische Marken als auch Zeilennummern in demselben Programm verwenden und alphanumerische Marken als Objekte jeder BASIC-Anweisung einsetzen, in der Zeilennummern erlaubt sind, außer als Objekt einer **IF...THEN**-Anweisung. In **IF...THEN**-Anweisungen erlaubt BASIC nur eine Zeilennummer, es sei denn, Sie verwenden ausdrücklich eine **GOTO**-Anweisung. Beispielsweise ist die folgende Anweisung korrekt:

```
IF A = 10 THEN 500
```

Ist das Objekt der **IF...THEN**-Anweisung jedoch eine Zeilenmarke, so wird eine **GOTO**-Anweisung erforderlich:

```
IF A = 10 THEN GOTO EinkommensDaten
```

Bei der Fehlerverfolgung gibt die Funktion **ERL** nur die letzte Zeilennummer vor einem Fehler an. Bei **RESUME** und **RESUME NEXT**-Anweisungen sind Zeilenmarken oder Zeilennummern nicht erforderlich. Für weitere Informationen lesen Sie bitte die Erklärungen für **ERL** und **RESUME**.

Hinweis Zeilennummern bestimmen nicht die Reihenfolge, in der Anweisungen in QuickBASIC ausgeführt werden. In dem folgenden Programm werden die Anweisungen von QuickBASIC in der Reihenfolge 100, 10, 5 ausgeführt:

```
100  Print "Die erste Zeile wird ausgeführt."
 10  Print "Die zweite Zeile wird ausgeführt."
  5  Print "Die dritte Zeile wird ausgeführt."
```

BASICA würde die Anweisungen in der Reihenfolge 5, 10, 100 ausführen.

1.2.2 BASIC-Anweisungen

Eine BASIC-Anweisung kann entweder "ausführbar" oder "nicht ausführbar" sein. Eine ausführbare Anweisung setzt den logischen Programmfluß fort, indem es dem Programm mitteilt, was als nächstes zu tun ist (wie lese Eingabe, schreibe Ausgabe, addiere zwei Zahlen, speichere das Ergebnis in einer Variablen, öffne eine Datei, verzweige zu einem anderen Programmteil usw.). Im Unterschied dazu führt eine nicht ausführbare Anweisung den logischen Programmfluß nicht fort. Nicht ausführbare Anweisungen erfüllen Aufgaben wie die Zuordnung von Speicherplatz für Variablen, Deklarieren und Definieren von Variablentypen und das Festlegen von Variablen, die allen Prozeduren einer Quelldatei zugänglich sein sollen.

Die folgenden BASIC-Anweisungen sind nicht ausführbar:

- **REM** oder ' (Kommentaranfang; schließt Metabefehle ein)
- **COMMON**
- **CONST**
- **DATA**
- **DECLARE**
- **DEF***Typ*
- **DIM** (nur statische Datenfelder)
- **OPTION BASE**
- **SHARED**
- **STATIC**
- **TYPE...END TYPE**

Ein "Kommentar" ist eine nicht ausführbare Anweisung. Er soll die Operationen und den Zweck eines Programms erklären. Ein einfaches Anführungszeichen (') oder die Anweisung **REM** leiten den Kommentar ein. Die folgenden Zeilen sind gleichbedeutend:

```
PRINT "Restschuld" : REM Drucke Rechnung.
PRINT "Restschuld"      'Drucke Rechnung.
```

In einer Zeile kann mehr als eine BASIC-Anweisung stehen; jede Anweisung muß jedoch von der vorhergehenden durch einen Doppelpunkt (:) getrennt werden.

```
FOR I=1 TO 5 : PRINT "Tag." : NEXT I
```

1.2.3 Die Zeilenlänge in BASIC

Wenn Sie Ihr Programm unter Verwendung des eingebauten QuickBASIC-Editors eingeben, sind Sie auf 256 Zeichen pro Zeile beschränkt. Der Editor erkennt den Unterstreichungsstrich (_) nicht als eine Zeilenfortsetzung an.

Wenn Sie Ihren eigenen Editor verwenden, können Sie den Unterstreichungsstrich als letztes Zeichen benutzen, um eine Programmzeile wie die folgende zu erstellen, die sich über mehr als eine physikalische Zeile erstreckt:

```
IF (TestZeich$ = " " OR TestZeich$ = ".") AND _
   ZeilNum < 23 AND NOT EOF(DateiNum) THEN
```

Wenn QuickBASIC Ihr Programm lädt, werden die Unterstreichungsstriche entfernt und die fortgesetzten Zeilen werden zu einer Zeile zusammengesetzt. In diesem Fall ist das Limit der Zeilenlänge aufgehoben. Unterstreichungsstriche dürfen nicht verwendet werden, um **DATA**- oder **REM**-Anweisungen fortzusetzen.



2 Datentypen

Dieses Kapitel enthält sieben Abschnitte, die die folgenden Informationen zu Datentypen, Konstanten und Variablen in BASIC bieten:

- Die elementaren Datentypen (Zeichenketten und numerische Datentypen)
- Den Aufbau von literalen und symbolischen Konstanten
- Die Regeln zur Benennung einer BASIC-Variablen, zur Festlegung des Typs einer Variablen, und wie BASIC Variablen Speicherplatz zuweist
- Die Bereichsregeln, denen BASIC folgt, um zu entscheiden, auf welches Objekt sich ein Variablenname bezieht
- Den Unterschied zwischen **$STATIC**- und **$DYNAMIC**-Datenfeldern
- Die Verwendung von automatischen und **STATIC**-Variablen in **SUB**- und **FUNCTION**-Prozeduren
- Die Umwandlungen von einem numerischen Typ in einen anderen, die BASIC während einer Berechnung durchführen kann

2.1 Datentypen

Jede BASIC-Variable hat einen Datentyp, der festlegt, was in der Variablen gespeichert werden kann. Es gibt zwei Kategorien in BASIC: Zeichenkettendaten und numerische Daten. Jede Kategorie schließt elementare Datentypen ein. Der nächste Abschnitt stellt die elementaren Datentypen kurz dar.

2.1.1 Elementare Datentypen – Zeichenketten

Nachfolgend die zwei Arten von Zeichenketten:

- Zeichenketten mit variabler Länge

  Eine Zeichenkette mit einer variablen Länge ist eine Folge von bis zu 32.767 Zeichen. Die Codes für diese Zeichen liegen im Bereich von 0 bis 127 des American Standard Codes for Information Interchange (ASCII) und von 128 bis 255 der Nicht-ASCII-Zeichen (siehe Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes").

- Zeichenketten mit fester Länge

  Eine Zeichenkette mit fester Länge enthält Zeichen bis zu einer deklarierten Höchstzahl. Zeichenketten mit fester Länge können nicht länger als 32.767 Zeichen sein. Sie enthalten, genauso wie Zeichenketten mit variabler Länge, Zeichen im Codebereich von 0 - 255.

2.1.2 Elementare Datentypen – Numerische

BASIC hat zwei Ganzzahltypen und zwei Gleitkommazahlentypen. Abbildung 2.1 zeigt die internen (Speicher-) Formate dieser vier Typen. Die folgende Liste faßt die numerischen Typen zusammen:

- Ganzzahl (2-Byte)

  Ganzzahlen werden als 16-Bit-Binärzahl im Wert von -32.768 bis +32.767 gespeichert.

- Lange (4-Byte-) Ganzzahl

  "Lange" Ganzzahlen werden als 32-Bit-Binärzahl im Wert von -2.147.483.648 bis +2.147.483.647 gespeichert.

- Gleitkommazahl (4-Byte) einfacher Genauigkeit

  Zahlen "einfacher Genauigkeit" sind bis auf 7 Dezimalstellen genau und können ungefähr im Bereich -3,402823E+38 bis -1,40129E-45 für negative Werte und 1,40129E-45 bis 3,402823E+38 für positive Werte liegen.

- Gleitkommazahl (8-Byte) doppelter Genauigkeit

  Zahlen "doppelter Genauigkeit" sind bis auf 15 oder 16 Stellen genau und haben einen erweiterten Bereich: -1,797693134862316E+308 bis -4,94065E-324 für negative Zahlen und 4,94065E-324 bis 1,797693134862316E+308 für positive Zahlen.

2.1.2.1 Ganzzahlen

Alle BASIC-Ganzzahlen werden als Zweierkomplementwerte dargestellt. Dies ist die übliche Darstellungsweise von Ganzzahlen in einem Computer. Ganzzahlen benutzen 16 Bits (2 Bytes), und lange Ganzzahlen benutzen 32 Bits (4 Bytes).

In der Zweierkomplementdarstellung werden positive Werte als einfache Binärzahlen dargestellt. Beispielsweise würde BASIC einen Ganzzahlwert von 4 als eine Reihenfolge der folgenden 16 Bits speichern:

```
0000000000000100
```

Negative Werte werden als das Zweierkomplement des entsprechenden positiven Wertes dargestellt. Um das Zweierkomplement (die Negation) des Ganzzahlwertes 4 zu bilden, nehmen Sie die obige Darstellung, und ändern Sie alle Nullen in Einsen und alle Einsen in Nullen:

```
1111111111111011
```

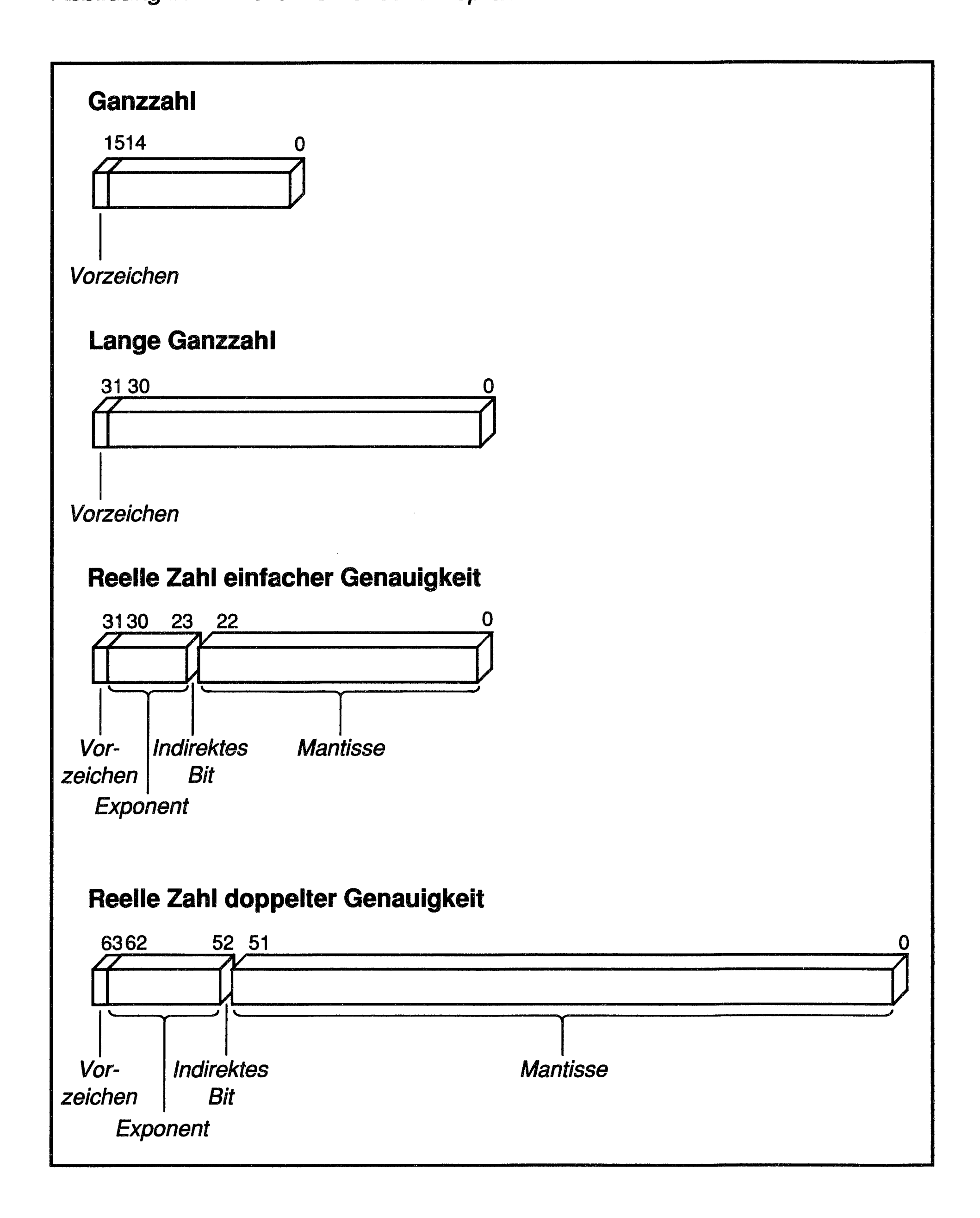
Addieren Sie dem Ergebnis dann eine Eins hinzu:

```
1111111111111100
```

Das Endergebnis zeigt, wie BASIC den Wert -4 als binäre Zahl darstellt.

Aufgrund der Art und Weise, wie ein Zweierkomplement gebildet wird, hat jede Kombination von Bits, die einen negativen Wert darstellt, eine 1 als äußerstes linkes Bit.

Abbildung 2.1 BASIC Numerische Repräsentation



2.1.2.2 Gleitkommazahlen

QuickBASIC verwendet Gleitkommazahlen im IEEE-Format, und nicht mehr das in früheren Versionen benutzte Microsoft-Binär-Format. (IEEE ist eine Abkürzung für Institute of Electrical und Electronics Engineers, Inc.) Das IEEE-Format gibt genauere Ergebnisse an und läßt die Benutzung eines 8087 oder 80287 mathematischen Koprozessors zu.

Gleitkommawerte werden in einem anderen Format als Ganzzahlen dargestellt. Jeder Gleitkommawert besteht aus drei Teilen: dem Vorzeichen, dem Exponenten und der Mantisse. Sie können sich dieses Format als Abwandlung der wissenschaftlichen Notation vorstellen. In der wissenschaftlichen Notation wird die Zahl 1000 als 1.0×10^3 dargestellt. Um Platz zu sparen, könnten Sie sich einfach den Exponenten 3 und die Mantisse 1.0 merken, und später den Wert durch Potenzieren von 10 mit dem Exponenten 3 und durch die Multiplikation mit der Mantisse rekonstruieren. Gleitkommanotation funktioniert, indem nur der Exponent und die Mantisse gespeichert werden. Der einzige Unterschied zur wissenschaftlichen Notation besteht darin, daß der Exponent eine Potenz von zwei, nicht eine Potenz von zehn, darstellt.

In einer Zahl einfacher Genauigkeit nimmt das Vorzeichen ein Bit in Anspruch, der Exponent acht Bits, und die Mantisse verwendet die verbleibenden dreiundzwanzig Bits und ein zusätzliches indirektes Bit. Werte doppelter Genauigkeit nehmen acht Bytes oder vierundsechzig Bits ein: ein Bit für das Vorzeichen, elf Bits für den Exponenten und ein indirektes Bit und zweiundfünfzig Bits für die Mantisse.

Das folgende Programm kann verwendet werden, um das interne Format von Werten einfacher Genauigkeit zu untersuchen:

```
' Zeige an, wie ein gegebener reeller Wert im
' Hauptspeicher abgelegt wird.
DEFINT A-Z
DECLARE FUNCTION MHex$ (X AS INTEGER)
DIM Bytes(3)
DO
   ' Hole den Wert.
   INPUT "Geben Sie den Wert ein (END zum Beenden): ", A$
   IF UCASE$(A$) = "END" THEN EXIT DO
   RealWert! = VAL(A$)
   ' Übertrage den reellen Wert in einen langen, ohne
   ' eines der Bits zu verändern.
   WieLang& = CVL(MKS$(RealWert!))
   ' Bilde eine Zeichenkette aus Hexzahlen und addiere
   ' führende Nullen hinzu.
   ZeichAus$ = HEX$(WieLang&)
   ZeichAus$ = STRING$(8 - LEN(ZeichAus$), "0") + ZeichAus$
```

```
   ' Sichere das Vorzeichenbit und eliminiere es dann,
   ' so daß es das Herausholen der Bytes nicht
   ' beeinflußt.
   VorzBit&=WieLang& AND &H80000000
   WieLang&=WieLang& AND &H7FFFFFFF
   ' Teile den reellen Wert in vier separate Bytes
   ' auf. Beachte, daß Dividieren die Bytes nach
   ' rechts schiebt, und daß AND unerwünschte Bits
   ' entfernt.
   FOR I = 0 TO 3
      Bytes(I) = WieLang& AND &HFF&
      WieLang&=WieLang& \ 256&
   NEXT I
   ' Zeige an, wie der Wert im Speicher erscheint.
   PRINT
   PRINT "Bytes im Speicher"
   PRINT "  Oben   Unten"
   FOR I = 1 to 7 STEP 2
      PRINT MID$(ZeichAus$, I, 2); " ";
   PRINT : PRINT
   ' Setze den Wert, der für das Vorzeichenbit
   ' angezeigt wird.
   Sign = ABS(VorzBit& <> 0)
   ' Der Exponent besteht aus den rechten sieben Bits
   ' von Byte 3 und dem äußerst linken Bit von Byte 2.
   ' Die Multiplikation mit 2 schiebt nach links und
   ' schafft Platz für das zusätzliche Bit von Byte 2.
   Exponent = Bytes(3) * 2 + Bytes(2) \ 128
   ' Der erste Teil der Mantisse besteht aus den
   ' rechten sieben Bits von Byte 2. Die OR-Operation
   ' stellt durch das Setzen des äußerst linken Bits
   ' sicher, daß das indirekte Bit angezeigt wird.
   Mant1 = (Bytes(2) OR &H80)
   PRINT "    Bit 31        Bits 30-23  Indirektes Bit"
   PRINT "    & Bits 22-0"
   PRINT "Vorzeichenbit  Exponentbits  Mantissenbits"
   PRINT TAB(7); Sign; TAB(23); MHex$(Exponent);
   PRINT TAB(41); MHex$(Mant1); MHex$(Bytes(1));
MHex$(Bytes(0))
   PRINT
LOOP
```

```
' MHex$ stellt sicher, daß wir immer zwei Hexzahlen
' erhalten.
FUNCTION MHex$ (X AS INTEGER) STATIC
   D$ = HEX$(X)
   IF LEN(D$) < 2 THEN D$ = "0" + D$
   MHex$ = D$
END FUNCTION
```

Ausgabe

```
Geben Sie den Wert ein (END zum Beenden): 4
Bytes im Speicher
  Oben    Unten
  40 80   00 00
    Bit 31        Bits 30-23  Indirektes Bit & Bits 22-0
Vorzeichenbit    Exponentbits        Mantissenbits
       0              81                   800000
```

Das Programm zeigt das Vorzeichen, den Exponenten und die Mantisse des Wertes einfacher Genauigkeit. Das indirekte Bit der Mantisse wird automatisch eingeschlossen. Alle Werte werden als hexadezimale Werte (zur Basis 16) geschrieben, die eine Kurzdarstellung von Binärzahlen darstellen. Jede Hexadezimalziffer stellt ein Muster aus vier Bits dar: 0 hexadezimal repräsentiert 0000 binär, 8 hexadezimal repräsentiert 1000 und F hexadezimal (eine Ziffer für den Wert 15 dezimal) repräsentiert 1111.

Wie die Ausgabe des Beispiels zeigt, wird der Wert 4 im Speicher als eine Serie von vier Bytes dargestellt:

```
40 80 00 00
```

Diese vier Bytes teilen sich in ein einziges Vorzeichenbit von 0, einen Ein-Byte-Exponenten von &H81 und eine Mantisse von &H800000 auf.

Der Exponentenwert von $H81 stellt einen voreingestellten Exponenten dar, nicht den wahren Exponenten. Bei einem voreingestellten Exponenten wird zu dem wahren Exponenten ein fester Wert addiert, und das Ergebnis wird als Teil der Zahl gespeichert. Für Werte mit einfacher Genauigkeit ist die Voreinstellung &H7F oder 127. Werte doppelter Genauigkeit verwenden eine Voreinstellung von &H3FF oder 1023. Die Verwendung eines voreingestellten Exponenten hat den Vorteil, daß kein Exponentenbit für die Darstellung des Vorzeichens benutzt werden muß. In der Ausgabe ist 4 eine Potenz von zwei, so daß der wahre Exponent 2 ist. Die Addition der Voreinstellung (&H7F) ergibt den gespeicherten Exponentenwert von &H81.

Im Speicher besteht die Mantisse für einen Wert von 4 aus lauter Nullen. Dies ist so, weil die Mantisse normalisiert ist, und das äußerst linke Bit vorgegeben ist. In einer normalisierten Mantisse wird die Mantisse mit 2 multipliziert (nach links geschoben) und der Exponent wird verkleinert, bis das äußerst linke Bit eine 1 ist. Indem führende Nullen aus der Mantisse entfernt werden, können mehr signifikante Bits gespeichert werden.

Weil die Mantisse immer normalisiert ist, ist das äußerste linke Bit immer 1. Es gibt daher keinen Grund, es als Teil der Zahl zu speichern. Deshalb speichert BASIC 23 Bits (Nummer 22 bis 0) für die Mantisse einer Zahl einfacher Genauigkeit, schließt aber ebenso ein vierundzwanzigstes indirektes Bit ein, das immer eine 1 ist.

Rechts von dem indirekten Bit gibt es einen "indirekten Binärpunkt" (analog zu einem Dezimalpunkt). Der Binärpunkt zeigt an, daß die Bits 22 bis 0 eigentlich ein binärer Bruch sind. Daher ist die Mantisse in dem Beispiel eigentlich eine einzige 1 (das indirekte Bit), gefolgt von einem Binärbruch mit 0 (Bits 22 bis 0). Die Mantisse von 1, multipliziert mit 2 und potenziert mit dem Exponenten, 2, ergibt 1×2^2 oder 4.

2.1.3 Benutzerdefinierte Datentypen

BASIC gibt Ihnen die Möglichkeit, unter Benutzung der Anweisung **TYPE** neue Datentypen zu definieren. Ein benutzerdefinierter Typ ist ein verbundener Typ, der sich aus elementaren BASIC-Typen zusammensetzt.

Zum Beispiel definiert die folgende **TYPE**-Anweisung einen Typ, `SymTabEingang`:

```
TYPE SymTabEingang
   Bezeichnung  AS STRING*40
   ZeilenNummer AS LONG
   Wert         AS LONG
END TYPE
```

Der neue Typ beinhaltet eine Zeichenkette mit fester Länge und zwei 4-Byte-Ganzzahlen. Ein benutzerdefinierter Typ belegt nur soviel Speicherplatz wie die Summe seiner Bestandteile. Ein `SymTabEingang` belegt 48 Bytes - 40 Bytes für die Zeichenkette mit fester Länge und 4 Bytes für jede der beiden 4-Byte-Ganzzahlen.

Außer Zeichenketten mit variabler Länge können Sie jeden der Standarddatentypen in einem benutzerdefinierten Typ benutzen: einfache und lange Ganzzahlen, Gleitkommazahlen mit einfacher und doppelter Genauigkeit und Zeichenketten mit fester Länge.

Hinweis Benutzerdefinierte Typen können keine Datenfelder oder Zeichenketten variabler Länge beinhalten.

2.2 Konstanten

Konstanten sind vordefinierte Werte, die sich während der Programmausführung nicht ändern. Es gibt zwei generelle Arten von Konstanten: literale Konstanten (wie Zahlen und Zeichenketten) und symbolische Konstanten.

2.2.1 Literale Konstanten

BASIC hat zwei Arten von literalen Konstanten: Zeichenkettenkonstanten und numerische Konstanten.

Eine Zeichenkettenkonstante ist eine Folge von bis zu 32.767 alphanumerischen Zeichen, die in doppelten Anführungszeichen stehen. Diese alphanumerischen Zeichen können alle Zeichen sein, deren ASCII-Codes im Bereich von 0 bis 255 liegen (mit Ausnahme des doppelten Anführungszeichens (") und der Wagenrücklauf-Zeilenvorschub-Sequenz). Dieser Bereich enthält sowohl die einfachen ASCII-Zeichen (0-127) als auch die erweiterten Zeichen (128-255). Gültige Zeichenkettenkonstanten sind beispielsweise:

```
"HALLO"
"DM 25.000.000"
"Anzahl der Angestellten"
```

Numerische Konstanten sind positive oder negative Zahlen. Es gibt vier Arten von numerischen Konstanten: kurze Ganzzahlen, lange Ganzzahlen, Festkomma- und Gleitkommazahlen. Die folgende Tabelle beschreibt diese Typen und zeigt Beispiele:

Typ	*Subtyp*	*Beschreibung*	*Beispiel*

Ganzzahl	Dezimal	Eine oder mehrere Dezimalziffer/n (0-9), mit wahlweise einem Vorzeichen (+ oder -). Der Bereich für kurze Dezimalkonstanten ist von -32.768 bis +32.767.	`68`
			`+407`
			`-1`

Ganzzahl	Hexadezimal	Eine oder mehrere Hexadezimalziffer/n (0-9, a-f oder A-F) mit vorangestelltem &H oder &h. Hexadezimale Konstanten können im Bereich zwischen &H0 und &HFFFF liegen.	`&H76` `&H32F`
Ganzzahl	Oktal	Eine oder mehrere Oktalziffer/n (0-7) mit vorangestelltem &O, &o oder &. Oktalkonstanten können im Bereich zwischen &o0 und &o177777 liegen.	`&o347` `&1234`

Typ	*Subtyp*	*Beschreibung*	*Beispiel*
Lange Ganzzahl	Dezimal	Eine oder mehrere Dezimalziffer/n (0-9) mit wahlweise einem Vorzeichen (+ oder -). Der Bereich für lange Dezimalkonstanten ist von -2.147.483.648 bis +2.147.483.647.	`95000000` `-400141`
Lange Ganzzahl	Hexadezimal	Eine oder mehrere Hexadezimalziffer/n (0-9, a-f oder A-F) mit vorangestelltem &H oder &h und nachgestelltem &. Lange hexadezimale Konstanten können im Bereich zwischen &h0& und &hFFFFFFFF& liegen.	`&HO&` `&H1AAAAA&`
Lange Ganzzahl	Oktal	Eine oder mehrere Oktalziffer/n (0-7) mit vorangestelltem &O, &o oder & und nachgestelltem &. Lange Oktalkonstanten können im Bereich zwischen &o0& und &o37777777777 liegen. Beachten Sie, daß hexadezimale und oktale Konstanten vorzeichenlos sind und keine Dezimalpunkte enthalten.	`&o347&` `&555577733&`
Festkomma		Positive oder negative reelle Zahlen, d. h. Zahlen, die einen Dezimalpunkt enthalten.	`9.0846`
Gleitkomma	einfache Genauigkeit	Positive oder negative Zahlen in Exponentialdarstellung (ähnlich der wissenschaftlichen Schreibweise). Eine Gleitkommakonstante einfacher Genauigkeit besteht aus einer Ganzzahl oder einer Festkommazahl mit wahlweise einem Vorzeichen (der Mantisse), gefolgt vom Buchstaben E und einer ganzen Zahl mit jeweils einem Vorzeichen (dem Exponenten). Der Wert der Konstanten ist das Ergebnis der Multiplikation der Mantisse mit dem Exponenten (einer Potenz von 10). Konstanten einfacher Genauigkeit liegen im Bereich von -3,37E +38 bis 3,37E + 38	`235.988E-7` `2359E6`

Typ	*Subtyp*	*Beschreibung*	*Beispiel*
Gleitkomma	doppelte Genauigkeit	Gleitkommakonstanten doppelter Genauigkeit haben dieselbe Form wie Gleitkommakonstanten einfacher Genauigkeit, aber werden mit dem Buchstaben D zur Anzeige des Exponenten, statt des E gekennzeichnet. Konstanten doppelter Genauigkeit liegen im Bereich von -1,67D + 308 bis 1,67D + 308	`4.35D-10`

Numerische Festkomma- und Gleitkommakonstanten können entweder von einfacher oder von doppelter Genauigkeit sein. Numerische Konstanten einfacher Genauigkeit werden in 7 Stellen Genauigkeit (plus dem Exponenten) gespeichert. Zahlen doppelter Genauigkeit werden mit 15 oder 16 Stellen Genauigkeit (plus dem Exponenten) gespeichert.

Eine Konstante einfacher Genauigkeit ist jede numerische Konstante, die eine der folgenden Eigenschaften hat:

- Exponentialdarstellung, angezeigt durch "E".
- Ein nachfolgendes Ausrufezeichen (!).
- Ein Wert mit einem Dezimalpunkt, der kein "D" im Exponenten oder ein nachfolgendes Nummernzeichen (#) hat und nicht länger als 15 Ziffern ist.
- Ein Wert ohne Dezimalpunkt, der weniger als 15 Ziffern lang ist, aber nicht als eine lange Ganzzahl dargestellt werden kann.

Eine Konstante doppelter Genauigkeit ist jede numerische Konstante, die eine der folgenden Eigenschaften hat:

- Exponentialdarstellung, angezeigt durch "D".
- Ein nachfolgendes Nummernzeichen (#).
- Ein Dezimalpunkt, kein "E" im Exponenten oder nachfolgendes Ausfrufezeichen (!) und mehr als 15 Ziffern.

Nachfolgend Beispiele für Festkommakonstanten:

Einfache Genauigkeit	*Doppelte Genauigkeit*
`46.8`	`345692811.5`
`-1.09E-6`	`-1.09432D-06`
`3489.0`	`3489.0#`
`22!`	`7654321.1234`

Hinweis Numerische Konstanten in BASIC dürfen keine Kommata enthalten.

2.2.2 Symbolische Konstanten

BASIC umfaßt symbolische Konstanten, die an Stelle von numerischen Werten oder Zeichenkettenwerten verwendet werden können. Der folgende Programmausschnitt deklariert zwei symbolische Konstanten und benutzt eine davon zur Dimensionierung eines Datenfeldes:

```
CONST MAXZEICHEN%=254, MAXPUF%=MAXZEICHEN%+1
DIM Puffer%(MAXPUF%)
```

Der Name einer symbolischen Konstanten unterliegt denselben Regeln wie ein BASIC-Variablenname. Sie können ein Typdeklarationszeichen (%, #, ! oder $) in den Namen einfügen, um den Typ kenntlich zu machen. Das Zeichen ist jedoch nicht Teil des Namens. Zum Beispiel können nach der folgenden Deklaration die Namen N!, N#, N$, N% und N& nicht als Variablennamen benutzt werden, da sie denselben Namen haben wie die Konstante:

```
CONST N = 45
```

Der Typ einer Konstanten wird entweder durch ein ausdrückliches Typdeklarationszeichen oder den Typ des Ausdrucks festgelegt. Symbolische Konstanten werden von **DEF***Typ*-Anweisungen nicht berührt.

Wenn Sie das Typdeklarationszeichen weglassen, wird der Konstanten ein auf dem Ausdruck basierender Typ zugewiesen. Zeichenketten liefern immer eine Zeichenkettenkonstante. Bei numerischen Ausdrücken wird der Ausdruck ausgewertet, und der Konstanten wird der einfachste Typ, der sie darstellen kann, zugeordnet. Wenn beispielsweise der Ausdruck ein Ergebnis angibt, das als Ganzzahl dargestellt werden kann, wird der Konstanten ein Ganzzahltyp zugewiesen.

In Abschnitt 2.4, "Geltungsbereich von Variablen und Konstanten", finden Sie Informationen zu dem Geltungsbereich von Konstanten. In Teil 2, "Nachschlageteil - Anweisungen und Funktionen", finden Sie in der Beschreibung zu **CONST** weitere Informationen darüber, wo und wann symbolische Konstanten benutzt werden können.

2.3 Variablen

Eine Variable ist ein Name, der sich auf ein bestimmtes Objekt bezieht – eine bestimmte Zahl, eine Zeichenkette oder einen Verbund. (Unter Verbund versteht man eine Variable, die als benutzerdefinierter Typ deklariert ist.) Einfache Variablen beziehen sich auf einzelne Zahlen, Zeichenketten oder Verbunde. Datenfeldvariablen beziehen sich auf eine Gruppe von Objekten desselben Typs.

Einer numerischer Variablen, entweder einfach oder Datenfeld, kann nur ein numerischer Wert zugeordnet werden (entweder Ganzzahl, lange Ganzzahl, einfache Genauigkeit oder doppelte Genauigkeit); einer Zeichenkettenvariablen kann nur ein Zeichenkettenwert zugeordnet werden. Sie können Verbundvariablen nur dann einander zuordnen, wenn beide Variablen denselben benutzerdefinierten Typ haben. Sie können jedoch immer einzelne Elemente eines Verbundes einer Variablen des entsprechenden Typs zuordnen.

Die folgende Liste zeigt einige Beipiele von Variablenzuweisungen:

- Einen konstanten Wert:

```
A = 4.5
```

- Den Wert einer anderen Zeichenkettenvariablen oder numerischen Variablen:

```
B$ = "Bahnhof"
A$ = B$
Gewinn = Nettoeinnahmen
```

- Den Wert eines Verbundelementes:

```
TYPE AngestellterRec
   Name AS STRING*25
   Vers AS STRING*9
END TYPE
DIM AktAngest AS AngestellterRec
.
.
.
AusVers$=AktAngest.Vers
```

- Der Wert von einer Verbundvariablen zu einer anderen Verbundvariablen desselben Typs:

```
TYPE DateiPuffer
   Name      AS STRING*25
   TaetigkNr AS INTEGER
END TYPE
DIM Puffer1 AS DateiPuffer, Puffer2 AS DateiPuffer
.
.
.
Puffer2=Puffer1
```

- Den Wert, den Sie durch die Kombination anderer Variablen, Konstanten und Operatoren erhalten:

```
CONST PI = 3.141593
Umwandlung = 180/PI
TempDatei$ = DateiSpez$ + ".BAK"
```

Weitere Informationen zu den zur Kombination von Variablen und Konstanten benutzten Operatoren finden Sie in Kapitel 3, "Ausdrücke und Operatoren".

In jedem Fall muß die Variable immer mit dem ihr zugeordneten Datentyp (numerisch oder Zeichenkette) übereinstimmen.

Hinweis Bevor einer Variablen ein Wert zugewiesen wird, wird (für numerische Variablen) der Wert Null oder (für Zeichenkettenvariablen) die Null-Zeichenkette angenommen. Alle Felder eines Verbundes, einschließlich Zeichenkettenfeldern, werden mit Null initialisiert.

2.3.1 Variablennamen

Ein Variablenname in BASIC kann bis zu 40 Zeichen enthalten. Die in einem Variablennamen erlaubten Zeichen sind Buchstaben, Zahlen, der Dezimalpunkt und die Typdeklarationszeichen (%, &, !, # oder $). Siehe Abschnitt 2.3.2.1, "Wie Sie einfache Variablen deklarieren". Das erste Zeichen eines Variablennamens muß ein Buchstabe sein. Wenn eine Variable mit **FN** beginnt, wird angenommen, daß sie der Aufruf einer **DEF FN**-Funktion ist.

Ein Variablenname kann kein reserviertes Wort sein; eingebettete reservierte Wörter sind jedoch erlaubt. Zum Beispiel ist die folgende Anweisung unzulässig, da **LOG** ein reserviertes Wort ist (BASIC unterscheidet nicht nach Groß- und Kleinbuchstaben).

```
log = 8
```

Folgende Anweisung ist jedoch zulässig:

```
TimeLog = 8
```

Reservierte Wörter umfassen sämtliche BASIC-Befehle, -Anweisungen, -Funktionsnamen und -Operatornamen. (Die vollständige Liste der reservierten Wörter finden Sie in Anhang B, "Reservierte Wörter in QuickBASIC".)

Hinweis Variablennamen müssen sich sowohl von den Namen der Prozeduren (**SUB** und **FUNCTION**) als auch von den Namen der symbolischen Konstanten (**CONST**) unterscheiden.

2.3.2 Variablentypen

Die beiden nächsten Abschnitte erläutern einfache Variablen (Variablen, die sich nur auf ein einzelnes Objekt beziehen) und Datenfeldvariablen (Variablen, die sich auf eine Gruppe von Objekten beziehen).

2.3.2.1 Wie Sie einfache Variablen deklarieren

Eine einfache Variable kann eine numerische, Zeichenketten- oder Verbundvariable sein. Sie können einfache Variablentypen auf eine der drei folgenden Arten angeben:

1. Hängen Sie eines der folgenden Typdeklarationszeichen an den Variablennamen an:

   % & ! # $

   Das Dollarzeichen ($) ist das Typdeklarationszeichen für Zeichenkettenvariablen mit variabler Länge. Es deklariert, daß die Variable eine Zeichenkette darstellt, und Sie ihr eine Zeichenkettenkonstante mit bis zu 32.767 Zeichen zuweisen können, wie im folgenden Beispiel:

   ```
   A$ = "Verkaufsbericht"
   ```

Numerische Variablen können Ganzzahlwerte (gekennzeichnet durch nachgestelltes "%"), lange Ganzzahlwerte (gekennzeichnet durch ein nachfolgendes "&"), Werte einfacher Genauigkeit (gekennzeichnet durch ein nachfolgendes "!") oder Werte doppelter Genauigkeit (gekennzeichnet durch ein nachgestelltes "#") deklarieren. Einfache Genauigkeit ist der Standardwert für Variablen ohne ein Typdeklarationszeichen.

Es gibt kein Typdeklarationszeichen für einen benutzerdefinierten Typ.

2. Deklarieren Sie Variablen in einer Deklaration in der Form

Deklariere Variablennamen **AS** *Typ*

wobei *Deklariere* entweder **DIM, COMMON, REDIM, SHARED** oder **STATIC** und *Typ* entweder **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder ein benutzerdefinierter Typ sein kann. Zum Beispiel deklariert die folgende Anweisung die Variable A als lange Ganzzahl:

```
DIM A AS LONG
```

Der **INTEGER**-Typname legt den Variablentyp als (2-Byte-) Ganzzahl fest.

Zeichenkettenvariablen, die in einer **AS STRING**-Klausel deklariert sind, können entweder Zeichenketten variabler Länge oder Zeichenketten mit fester Länge sein. Zeichenketten mit variabler Länge sind "dehnbar". Ihre Länge hängt von der Länge der Zeichenkette ab, die ihnen zugeordnet wird. Zeichenketten mit fester Länge haben eine konstante Länge, die durch das Hinzufügen von **Zahl* an die **AS STRING**-Klausel festgelegt wird. Diese *Zahl* ist die Länge der Zeichenkette in Bytes. Zum Beispiel:

```
' Zeichenkette1 kann eine variable Länge haben:
DIM Zeichenkette1 AS STRING
' Zeichenkette2 hat eine feste Länge von 7 Bytes:
DIM Zeichenkette2 AS STRING*7
Zeichenkette1 = "1234567890"
Zeichenkette2 = "1234567890"
PRINT Zeichenkette1
PRINT Zeichenkette2
```

Ausgabe

```
1234567890
1234567
```

Weitere Informationen zu Zeichenketten mit fester Länge und Zeichenketten mit variabler Länge finden Sie in *Programmieren in BASIC: Ausgewählte Themen,* Kapitel 4, "Die Verarbeitung von Zeichenketten".

Sie deklarieren Verbundvariablen, indem Sie den Namen des benutzerdefinierten Typs in der **AS**-Klausel benutzen:

```
TYPE InventurPosten
   Beschreibung AS STRING*40
   Nummer AS STRING*10
   Menge AS LONG
   Bestell AS LONG
END TYPE
DIM AktuellPosten AS InventurPosten, VorhergPosten AS_
    InventurPosten
```

Um Bezug auf einzelne Elemente der neuen Variablen zu nehmen, benutzen Sie folgendes Format: *Variablenname.Elementname.* Zum Beispiel:

```
Beschr$ ="Ergonomischer Schreibtischstuhl"
IF AktuellPosten.Beschreibung = Beschr$ THEN
   PRINT AktuellPosten.Nummer; AktuellPosten.Menge
END IF
```

Hinweis Wenn Sie eine Variable mit der **AS**-Klausel deklarieren, muß für jede Deklaration der Variablen eine **AS**-Klausel benutzt werden. Im folgenden Programmausschnitt ist die **AS**-Klausel in der **COMMON**-Anweisung verlangt, da **AS** in der **DIM**-Anweisung benutzt wurde:

```
CONST MAXANGESTELLTE=250
DIM AngestNamen(MAXANGESTELLTE) AS STRING
COMMON AngestNamen() AS STRING
.
.
.
```

3. Benutzen Sie die BASIC-Anweisungen **DEFINT, DEFLNG, DEFSTR, DEFSNG** und **DEFDBL**, um die Typen für bestimmte Variablennamen zu deklarieren. Durch die Verwendung einer dieser **DEF***Typ*-Anweisungen können Sie festlegen, daß alle Variablen, die mit einem gegebenen Buchstaben oder Buchstabenbereich anfangen, zu einem bestimmten Variablentyp gehören, ohne ein nachfolgendes Deklarationszeichen angeben zu müssen.

DEF*Typ*-Anweisungen berühren nur die Variablennamen in dem Modul, in dem sie vorkommen. Weitere Informationen zu der Anweisung **DEF***Typ* finden Sie in Teil 2 dieses Buches, "Nachschlageteil - Anweisungen und Funktionen".

Die Typdeklarationszeichen für Variablennamen, die in **AS**-*Typ*-Deklarationen akzeptierten Typnamen und den Speicherplatz (in Bytes), der für jeden Variablentyp zur Speicherung des Variablenwertes benötigt wird, finden Sie in der folgenden Tabelle.

Deklarations-zeichen	*AS-Typname*	*Variablentyp*	*Erforderliche Bytes*
%	**INTEGER**	Ganzzahl	2
&	**LONG**	Lange Ganzzahl	4
!	**SINGLE**	Einfache Genauigkeit	4
#	**DOUBLE**	Doppelte Genauigkeit	8
$	**STRING**	Zeichenkette variabler Länge	Benötigt 4 Bytes für Beschreiber, 1 Byte für jedes Zeichen in der Zeichenkette
$	**STRING***_Zahl_	Zeichenkette fester Länge	Benötigt *Zahl* Bytes
(Ohne Anhang für Verbund-variablen)	Benutzerdefiniert		Benötigt soviele Bytes, wie die individuellen Elemente erfordern

2.3.2.2 Datenfeldvariablen

Ein Datenfeld ist eine Gruppe von Objekten, die sich auf denselben Variablennamen beziehen. Die einzelnen Werte in einem Datenfeld sind die Elemente. Datenfeldelemente sind also Variablen und können in jeder BASIC-Anweisung oder -Funktion, die mit Variablen arbeitet, verwendet werden. Sie "dimensionieren" ein Datenfeld, wenn Sie es zum ersten Mal benutzen, oder wenn Sie Namen, Typ und Anzahl der Elemente im Datenfeld festlegen.

Auf jedes Element in einem Datenfeld wird mit einer Datenfeldvariablen Bezug genommen, die mit einer Ganzzahl oder einem ganzzahligen Ausdruck indiziert ist. (Sie können nichtganzzahlige Ausdrücke als Datenfeldindizierung verwenden, jedoch werden diese auf ganzzahlige Werte gerundet.) Ein Datenfeldvariablenname hat soviele Indizes, wie das Datenfeld Dimensionen hat. Beispielsweise bezieht sich `V(10)` auf einen Wert in einem eindimensionalen Datenfeld, während `T$(1,4)` sich auf einen zweidimensionalen Wert in einem Zeichenketten-Datenfeld bezieht.

Der vorgegebene obere Indexwert für jede Datenfelddimension ist 10. Der maximale Wert kann durch die Anweisung **DIM** geändert werden. (Weitere Informationen zur **DIM**-Anweisung finden Sie auf den Nachschlageseiten in Teil 2, "Nachschlageteil - Anweisungen und Funktionen".) Die maximale Anzahl der Dimensionen für ein Datenfeld ist 60. Die maximale Anzahl der Elemente pro Dimensionen ist 32.768.

Sie können Datenfelder mit jedem einfachen Variablentyp, einschließlich der Verbunde, deklarieren. Um ein Datenfeld mit Verbunden zu deklarieren, deklarieren Sie zunächst den Datentyp in einer **TYPE**-Anweisung und dimensionieren Sie dann das Datenfeld.

```
TYPE BaumKnoten
   LinksPtr AS INTEGER
   RechtsPtr AS INTEGER
   DatenFeld AS STRING*20
END TYPE
DIM Baum(500) AS BaumKnoten
```

Jedes Element des Datenfeldes `Baum` ist ein Verbund des Typs `BaumKnoten`. Um ein einzelnes Element eines Verbundes in einem Datenfeld anzusprechen, benutzen Sie die Standard-Punktdarstellung (*variablenname.elementname*):

```
CONST MAXANGESTELLTE=500
TYPE AngestellteRec
   Name AS STRING*25
   Vers AS STRING*9
END TYPE
DIM Angestellte(MAXANGESTELLTE) AS AngestellteRec
.
.
.
PRINT Angestellte(I).Name;" "; Angestellte(I).Vers
```

Hinweis Datenfeldnamen werden von einfachen Variablennamen unterschieden. Die Datenfeldvariable `T` und die einfache Variable `T` im folgenden Beispiel sind zwei unterschiedliche Variablen:

```
DIM T(11)
T = 2          'Dies ist eine einfache Variable
T(0) = 1       'T(0) ist ein Datenfeld-Element
FOR I% = 0 TO 10
    T(I% + 1) = T * T(I%)
NEXT
```

Die Benutzung gleicher Namen für Datenfelder und einfache Variablen erhöht die Wahrscheinlichkeit von Fehlern in Ihrem Programm.

Datenfeldelemente benötigen, wie einfache Variablen, je nach Variablentyp eine bestimmte Speicherkapazität. Informationen über die Speicheranforderungen zur Speicherung von Datenfeldern finden Sie in der Tabelle in Abschnitt 2.3.2.1.

Um die Gesamtspeicherkapazität zu ermitteln, die ein Datenfeld benötigt, multiplizieren Sie die Anzahl der Elemente mit den Bytes pro Element, die der Datenfeldtyp erfordert. Betrachten Sie zum Beispiel die folgenden zwei Datenfelder:

```
DIM Array1 (1 TO 100) AS INTEGER
DIM Array2 (-5 TO 5)
```

Das erste Datenfeld, `Array1`, hat 100 ganzzahlige Elemente, somit beträgt die benötigte Speicherkapazität 200 Bytes. Das zweite Datenfeld, `Array2`, hat 11 Elemente doppelter Genauigkeit und benötigt somit 88 Bytes Speicherplatz. Da BASIC neben den Datenfeldwerten noch Informationen zu dem Datenfeld speichern muß, benötigen Datenfelder somit mehr Speicherplatz als zum Abspeichern der Werte nötig ist.

2.3.3 Speicherzuweisung für Variablen

BASIC speichert verschiedene Variablenarten in verschiedenen Speicherbereichen. Sie müssen sich nur dann darüber Gedanken machen, wo Variablen gespeichert werden, wenn Sie in verschiedenen Programmiersprachen programmieren, oder wenn Sie eine der folgenden BASIC-Anweisungen oder -Funktionen verwenden:

- **CALL, CALLS** (Nicht-BASIC-Prozeduren)
- **DECLARE** (Nicht-BASIC-Prozeduren)
- **SADD**
- **SETMEM**
- **VARPTR**
- **VARSEG**
- **VARPTR$**

BASIC speichert Variablen entweder in einem Bereich namens **DGROUP** oder als weite (far) Objekte. (**DGROUP** ist der Name des Standarddatensegmentes, jenes Segmentes, auf das bei der Verwendung von **DEF SEG** ohne Adresse Bezug genommen wird.) Auf Variablen, die in **DGROUP** gespeichert sind, kann mit kurzen (near) Adressen oder Zeigern (Pointer) Bezug genommen werden. Eine kurze Adresse besteht aus einem einzigen Wert oder Offset vom Beginn eines Segmentes oder Speicherblockes. Auf weite Objekte wird mit der Verwendung langer Adressen oder Zeiger Bezug genommen. Eine lange Adresse besteht aus zwei Teilen: der Startadresse des Segmentes oder Speicherblockes und dem Offset innerhalb des Segmentes. In den Beschreibungen zu den oben aufgelisteten Anweisungen finden Sie weitere Informationen darüber, wie Sie lange Zeiger (Far Pointer) erhalten und verwenden.

Ob eine Variable in **DGROUP** oder als weites Objekt gespeichert wird, hängt zunächst davon ab, ob sie eine einfache Variable oder ein Datenfeld ist. Alle einfachen Variablen werden in **DGROUP** gespeichert. Das Speichern von Datenfeldern ist ein bißchen komplizierter und leicht unterschiedlich für Programme, die als **.EXE**-Dateien bzw. innerhalb der QuickBASIC-Umgebung laufen.

In Programmen, die als **.EXE**-Dateien laufen, werden Datenfeldvariablen wie folgt gespeichert:

- Alle **$STATIC**-Datenfelder werden in **DGROUP** gespeichert, und es kann mit kurzen Adressen auf sie Bezug genommen werden.
- Alle **$DYNAMIC**-Datenfelder von Zeichenketten variabler Länge werden ebenfalls in **DGROUP** gespeichert, und es kann ebenfalls mit kurzen Adressen auf sie Bezug genommen werden.
- Alle anderen **$DYNAMIC**-Datenfelder werden als weite Objekte gespeichert und erfordern lange Adressen.

Eine Beschreibung von **$STATIC**- und **$DYNAMIC**-Datenfeldern finden Sie in Abschnitt 2.5.

In Programmen, die innerhalb der QuickBASIC-Umgebung laufen, folgt die Speicherung von Datenfeldvariablen folgenden Regeln:

- Alle **$STATIC**-Datenfelder in **COMMON** werden in **DGROUP** gespeichert, und es kann mit kurzen Adressen auf sie Bezug genommen werden.
- Alle Datenfelder von Zeichenketten variabler Länge werden ebenfalls in **DGROUP** gespeichert, und es kann ebenfalls mit kurzen Adressen auf sie Bezug genommen werden.
- Alle anderen Datenfelder werden als weite Objekte gespeichert und erfordern lange Adressen.

Weil BASIC versucht, eine möglichst effiziente Verwendung des Speichers zu gewährleisten, kann es verschiedene Ursachen für die Bewegung von Variablen im Speicher geben:

- Der Bezug auf ein Zeichenkettenliteral oder einen Zeichenkettenausdruck
- Der Aufruf einer **DEF FN** oder **FUNCTION**
- Die Verwendung einer BASIC-Funktion, die sich auf eine Zeichenkette oder den Hauptspeicher bezieht
- Ein Bezug auf ein implizit definiertes Datenfeld

Da BASIC-Variablen sich verschieben können, sollten Sie die Ergebnisse von **VARPTR**, **VARSEG**, **VARPTR$** oder **SADD** sofort nach dem Funktionsaufruf verwenden.

2.4 Geltungsbereich von Variablen und Konstanten

Wenn eine Variable erscheint, ermittelt BASIC anhand einer Reihe von Regeln, auf welches Objekt die Variable verweist. Diese Regeln legen den Geltungsbereich von Variablen fest – der Bereich von Anweisungen, in dem eine Variable definiert ist.

BASICA hat sehr einfache Geltungsbereichsregeln: Eine Variable wird eingeführt, wenn Sie sie zum erstenmal benutzen, und bleibt erhalten, bis das Programm endet. Der Geltungsbereich erstreckt sich vom ersten Gebrauch einer Variablen bis zum Ende des Programms.

In QuickBASIC können Sie den Geltungsbereich von Variablen und symbolischen Konstanten kontrollieren. Die Kontrolle über die Geltungsbereiche von Variablen und Konstanten hilft Ihnen, kurze, wohldefinierte Unterprogramme und **FUNCTION**-Prozeduren zu schreiben, die sich nicht gegenseitig stören. Sie können auch einige Variablen für alle Prozeduren in einem Modul verfügbar machen und auf diesem Weg wichtige Datenstrukturen mit Prozeduren teilen.

Variablen und Konstanten können einen von zwei Geltungsbereichen haben: global oder lokal. Einmal global deklarierte Variablen können an beliebiger Stelle in einem Modul benutzt werden, um einem einzelnen Objekt zugeordnet zu werden. Lokale Variablen sind lokal zu einem bestimmten Teil eines Moduls, dem Modul-Ebenen-Code oder einer der Prozeduren. Außerdem können Variablen so geteilt werden, daß sie weder ganz global noch ganz lokal sind.

Der folgende Abschnitt beschreibt den globalen und lokalen Geltungsbereich und geteilte Variablen. Das folgende Programmgerüst wird in dieser Beschreibung benutzt. Das Programm, ein Hauptprogramm und zwei Prozeduren, ersetzt Folgen von Leerzeichen in einer Datei mit Tab-Zeichen, ein erster einfacher Schritt zur Komprimierung einer Datei.

```
' ENTAB.BAS
'
'Programmgerüst, welches Folgen von Leerschritten in
'einer Datei durch Tabulatoren ersetzt
DEFINT a-z
DECLARE FUNCTION DiesIstEinTab (Spalte AS INTEGER)
CONST MAXZEILE=255, TABABST=8
CONST NEIN=0, JA=NOT NEIN
DIM SHARED TabStops (MAXZEILE)
.
.
.
' Setze die Tabulator-Postionen (verwendet das globale
' Datenfeld TabStops).
```

```
CALL SetzeTabPos
.
.
.
      IF DiesIstEinTab(AktuelleSpalte) THEN
         PRINT CHR$(9);
         LetzteSpalte=AktuelleSpalte
      END IF
.
.
.
'---------SUB SetzeTabPos-----------------------
' Setze die Tabulatorposition in dem Datenfeld
' TabStops
'
SUB SetzeTabPos STATIC
   FOR I=1 TO MAXZEILE
      TabStops(I)=((I MOD TABABST)=1)
   NEXT I
END SUB
'-------------FUNCTION DiesIstEinTab------------
' Beantwortet die Frage, "Ist dieses eine Tabulator-
' Postion?"
'
FUNCTION DiesIstEinTab(LetzteSpalte AS INTEGER) STATIC
   IF LetzteSpalte>MAXZEILE THEN
      DiesIstEinTab=JA
   ELSE
      DiesIstEinTab=TabStops(LetzteSpalte)
   END IF
END FUNCTION
```

2.4.1 Globale Variablen und Konstanten

Sowohl Variablen als auch symbolische Konstanten können in einem BASIC-Programm global sein. Eine globale Variable oder globale Konstante ist für das gesamte Modul definiert. Für eine Variable besteht die einzige Möglichkeit, diese global zu machen, darin, sie im Modul-Ebenen-Code mit dem **SHARED**-Attribut in einer **DIM-**, **REDIM-** oder **COMMON**-Anweisung zu deklarieren. Eine symbolische Konstante ist eine globale Konstante, falls sie im Modul-Ebenen-Code mit einer **CONST**-Anweisung deklariert wurde.

Im Beispiel-Programm ist das Datenfeld `TabStops` eine globale Variable. Da `TabStops` in der **DIM**-Anweisung mit dem Schlüsselwort **SHARED** deklariert ist, wird sie von allen Prozeduren im Modul geteilt. Beachten Sie, daß beide Prozeduren des Programms (`DiesIstEinTab` und `SetzeTabPos`) `TabStops` benutzen, indem sie darauf Bezug nehmen. Globale Variablen erfordern keine zusätzlichen Deklarationen, um in den Prozeduren eines Moduls benutzt zu werden. Auch die symbolischen Konstanten `MAXZEILE` und `TABABST` sind globale Konstanten. Beachten Sie, daß Sie bei der Benutzung des Namens einer globalen Variablen oder Konstanten in einer Prozedur auf die globale Variable und nicht auf die lokale Variable des gleichen Namens Bezug nehmen.

Hinweis Die **SHARED**-Anweisung erlaubt es Prozeduren, Variablen mit dem Modul-Ebenen-Code zu teilen. Dies ist nicht identisch mit der Globalisierung einer Variablen. In Abschnitt 2.4.3, "Das Teilen von Variablen", finden Sie weitere Informationen.

2.4.2 Lokale Variablen und Konstanten

Eine lokale Variable oder Konstante existiert nur innerhalb einer Prozedur oder des Modul-Ebenen-Codes. Wenn der Name einer lokalen Variablen in einer anderen Prozedur eines Moduls verwendet wird, stellt der Name eine andere Variable dar und bezieht sich auf ein anderes Objekt.

Das obige Beispielprogramm enthält viele globale Variablen. Die Variablen `LetzteSpalte` und `AktuelleSpalte` sind lokale Variablen des Modul-Ebenen-Codes. Die Variable `I` ist lokal, bezogen auf das Unterprogramm `SetzeTabPos`. Die Variable `LetzteSpalte` in der Parameterliste der Funktion `DiesIstEinTab` kann als lokale Variable betrachtet werden, da sie ein formaler Parameter ist. (Erinnern Sie sich, daß man unter einem formalen Parameter einen Platzhalter für den wirklichen, der Prozedur übergebenen, Parameter versteht. Weitere Informationen hierzu finden Sie in Kapitel 4.3 "Referenzübergabe und Wertübergabe".)

Die einfachste Denkweise hinsichtlich einer lokalen Variablen besteht darin, sie als nicht-global zu verstehen. Jede Variable, die im Modul-Ebenen-Code oder in einer Prozedur erscheint, ist lokal, sofern sie nicht in einer **DIM**-, **REDIM**- oder **COMMON**-Anweisung mit dem Attribut **SHARED** deklariert ist. (Es gibt eine Ausnahme. Siehe Kapitel 2.4.3, "Das Teilen von Variablen", unten.) Selbst wenn eine Variable in einer dieser Anweisungen steht, können Sie immer noch eine lokale Variable mit demselben Namen in einer Prozedur benutzen, indem Sie die Variable in einer **STATIC**-Anweisung deklarieren.

Wenn das Beispiel-Programm eine **DIM SHARED**-Anweisung enthalten würde, die `I` als globale Variable im Hauptprogramm deklariert, könnten Sie `I` in `SetzeTabPos` zu einer lokalen Variablen machen, indem Sie eine **STATIC**-Anweisung direkt hinter der **SUB**-Anweisung anfügen:

```
SUB SetzeTabPos STATIC
STATIC I
.
.
.
```

Jede symbolische Konstante, die in einer **SUB**- oder **FUNCTION**-Prozedur deklariert wird, ist eine lokale Konstante. Zum Beispiel ist in dem folgenden Fragment `ENDLIST` eine symbolische Konstante, die nur in der Funktion `FindeElement` existiert:

```
FUNCTION FindeElement (X())
CONST ENDLIST = - 32767
.
.
.
END FUNCTION
```

Hinweis Eine **STATIC**-Anweisung deklariert eine Variable nicht nur als lokale Variable: Ebenso weist sie den Compiler an, den Wert der Variablen zwischen den Prozeduraufrufen zu speichern. Benutzen Sie **STATIC**-Anweisungen nicht in rekursiven Prozeduren, wenn Sie die Variablenwerte zwischen den Aufrufen nicht speichern wollen. In Abschnitt 2.6, "Automatische und STATIC-Variablen", finden Sie weitere Informationen.

2.4.3 Das Teilen von Variablen

Sie können Variablen zwischen Komponenten eines Moduls teilen, ohne die Variable unter Verwendung der **SHARED**-Anweisung global zu machen. Um beispielsweise `TabStops` zu teilen, ohne es zu einer globalen Variablen zu machen, müßten Sie das **SHARED**-Attribut im Modul-Ebenen-Code entfernen und den beiden Prozeduren **SHARED**-Anweisungen hinzufügen.

```
DIM TabStops (MAXZEILE)
.
.
.
SUB SetzeTabPos STATIC
SHARED TabStops ()
.
.
.
FUNCTION DiesIstEinTab (LetzteSpalte AS INTEGER) STATIC
SHARED TabStops ()
.
.
.
```

Die **SHARED**-Anweisungen zeigen an, daß sich der Name `TabStops` in beiden Prozeduren auf dieselbe, im Modul-Ebenen-Code definierte, Variable bezieht.

2.4.4 DEF FN-Funktionen

Die **DEF FN**-Funktionen sind eine Ausnahme in den BASIC-Geltungsbereichsregeln. Jede Variable in einer **DEF FN**-Funktion, die nicht in der Parameterliste steht, ist Teil des Modul-Ebenen-Codes. Um eine Variable in Bezug auf **DEF FN** lokal zu machen, müssen Sie die Variable in einer **STATIC**-Anweisung deklarieren. Die **STATIC**-Anweisung in der folgenden **DEF FN**-Funktion macht die Variable `I` lokal:

```
CONST NEIN = 0, JA = NOT NEIN
DEF FNIstDaEineNull (A$)
STATIC I
   FOR I=1 TO LEN(A$)
      IF UCASE$ (MID$(A$, I, 1)) = "Z" THEN
         FNIstDaEineNull = JA
         EXIT DEF
      NEXT I
   FNIstDaEineNull = NEIN
END DEF
```

2.4.5 Zusammenfassung der Geltungsbereichsregeln

Die folgende Liste faßt die BASIC-Geltungsbereichsregeln zusammen:

- Eine Variable, die in einer **DIM-**, **REDIM-** oder **COMMON**-Anweisung mit dem **SHARED**-Attribut deklariert wurde, ist eine globale Variable. Jede **SUB**- oder **FUNCTION**-Prozedur kann auf die Variable Bezug nehmen.
- Eine symbolische Konstante ist eine globale Konstante, wenn sie in einer **CONST**-Anweisung im Modul-Ebenen-Code deklariert wurde. Symbolische Konstanten, die in einer **SUB** oder **FUNCTION** deklariert werden, sind lokal.
- Eine Variable ist eine lokale Variable, wenn sie in einer Prozedur erscheint und nicht als globale Variable deklariert wurde. Sie können den Namen einer globalen Variablen in einer Prozedur als lokale Variable verwenden, indem Sie diese in der Prozedur mit der **STATIC**-Anweisung deklarieren oder als formalen Parameter benutzen.
- Die **SHARED**-Anweisung ermöglicht es Ihnen, eine Variable mit dem Modul-Ebenen-Code und anderen Prozeduren - mit äquivalenten **SHARED**-Anweisungen - zu teilen, ohne die Variable global zu machen.
- Alle Variablen in einer **DEF FN**-Funktion sind Teil des Modul-Ebenen-Codes, solange sie nicht ausdrücklich mit Hilfe der **STATIC**-Anweisung lokal gemacht wurden.

2.5 $STATIC- und $DYNAMIC-Datenfelder

Sie können eine bessere Kontrolle über die Benutzung des von Ihrem Programm benötigten Speicherplatzes erhalten, wenn Sie den Zeitpunkt, zu dem Speicherplatz für Datenfelder geschaffen wird, kontrollieren können. Speicherplatz für Datenfelder kann entweder während des Kompilierens oder während der Ausführung des Programms geschaffen werden. Datenfelder, denen Speicherplatz während des Kompilierens zugewiesen wird, nennt man **$STATIC**-Datenfelder. **$DYNAMIC**-Datenfeldern wird Speicherplatz während der Ausführung zugewiesen. Der Speicherplatz, der von einem **$DYNAMIC**-Datenfeld benutzt wird, kann, während das Programm nicht läuft, entfernt werden, um Speicherplatz für andere Zwecke zu schaffen. Unter den Beschreibungen zu **DIM, ERASE** und **REDIM** in Teil 2, "Nachschlageteil – Anweisungen und Funktionen", finden Sie weitere Informationen zur Manipulierung von **$DYNAMIC**-Datenfeldern.

In der Deklaration wird festgelegt, ob es sich um ein **$STATIC**- oder ein **$DYNAMIC**-Datenfeld handeln soll. Standardmäßig sind alle Datenfelder, die mit konstanten Indizes dimensioniert sind oder implizit dimensioniert sind, **$STATIC**-Datenfelder. Datenfelder, die mit variablen Indizes dimensioniert sind oder zunächst in einer **COMMON**-Anweisung deklariert werden, sind **$DYNAMIC**-Datenfelder.

Sie können auch die Metabefehle **$STATIC** und **$DYNAMIC** verwenden, um zu kontrollieren, wie Datenfeldspeicherplatz zugewiesen wird. Siehe Anhang C, "Metabefehle", für weitere Informationen

Hinweis In einigen Fällen können Sie für Zeichenketten mehr Speicherplatz schaffen, indem Sie ein **$STATIC**-Datenfeld durch ein **$DYNAMIC**-Datenfeld ersetzen. In Programmen, die als **.EXE**-Dateien laufen, wird der Speicherplatz für **$STATIC**-Datenfelder in **DGROUP** zugewiesen – ein Bereich, in dem Zeichenketten gespeichert werden. Im Unterschied zu Datenfeldern von Zeichenketten variabler Länge, nehmen **$DYNAMIC**-Datenfelder keinen Speicherplatz in **DGROUP** ein – sie werden als weite Objekte gespeichert und benötigen lange Adressen.

2.6 Automatische und STATIC-Variablen

BASIC-Prozeduren können sowohl automatische als auch **STATIC**-Variablen benutzen. Automatische Variablen werden am Anfang eines jeden Aufrufs einer **FUNCTION** oder **SUB** initialisiert; **STATIC**-Variablen behalten ihren Wert bei.

Sie können festlegen, ob die Voreinstellung der Variablen automatisch oder **STATIC** ist, indem Sie das **STATIC**-Schlüsselwort in der **SUB**- oder **FUNCTION**-Anweisung verwenden oder auslassen. Wenn Sie **STATIC** auslassen, ist die Voreinstellung der Variablen "automatisch". Wenn Sie **STATIC** verwenden, ist die Voreinstellung für alle Variablen in der Prozedur **STATIC**; die Werte der Variablen werden zwischen den Prozeduraufrufen gespeichert.

Um nur einige Variablen in einer Prozedur **STATIC** zu machen, können Sie die Voreinstellung "automatisch" lassen (unter Auslassung von **STATIC** in der **SUB**- oder **FUNCTION**-Anweisung) und die **STATIC**-Anweisung benutzen.

Das folgende Programm benutzt eine **FUNCTION**, die zwei **STATIC**-Variablen enthält: `Start%` und `SpeicherZk$`. Die anderen Variablen sind automatisch. Die **FUNCTION** nimmt eine Zeichenkette und gibt ein Token – eine Folge von Zeichen – aus, bis das Ende der Zeichenkette erreicht ist. Beim ersten Aufruf schreibt `ZkTok$` eine lokale Kopie der Zeichenkette `Src$` in die **STATIC**-Variable `SpeicherZk$`. Nach dem ersten Aufruf gibt `ZkTok$` zusätzliche Token der Zeichenkette aus. Dabei verwendet es die **STATIC**-Variablen `Start%`, um sich zu erinnern, wo es gestartet ist. Alle anderen Variablen (`BegPos%`, `Ln%`, etc.) sind automatisch.

```
DECLARE FUNCTION ZkTok$(Quell$,Begrenzer$)
LINE INPUT "Zeichenkette eingeben: ";P$
' Definiere Zeichen, die die Token begrenzen.
Begrenzer$=" ,;:().?"+CHR$(9)+CHR$(34)
' Rufe ZkTok$ mit der zu tokenisierenden Zeichenkette
' auf.
Token$=ZkTok$(P$,Begrenzer$)
WHILE Token$<>""
   PRINT Token$
   ' Rufe ZkTok$ mit Null-Zeichenkette auf, so daß
   ' klar ist, daß dies nicht der erste Aufruf ist.
Token$=ZkTok$("",Begrenzer$)
WEND
FUNCTION ZkTok$(Srce$,Begr$)
STATIC Start%, SpeicherZk$
   ' Wenn erster Aufruf, Kopie der Zeichenkette
   ' erstellen.
   IF Srce$<>"" THEN
      Start%=1 : SpeicherZk$=Srce$
   END IF
   BegPos%=Start% : Ln%=LEN(SpeicherZk$)
   ' Anfang eines Token (Zeichen, das kein Begrenzer
   ' ist) suchen.
   WHILE BegPos%<=Ln% AND_
         INSTR(Begr$,MID$(SpeicherZk$,BegPos%,1))<>0
      BegPos% = BegPos% + 1
   WEND
   ' Überprüfung, ob Anfang des Token gefunden.
   IF BegPos% > Ln% THEN
      ZkTok$="" : EXIT FUNCTION
   END IF
   ' Finde das Ende des Token.
   EndPos%=BegPos%
   WHILE EndPos% <= Ln% AND_
         INSTR(Begr$,MID$(SpeicherZk$,EndPos%,1))=0
      EndPos%=EndPos%+1
   WEND
   ZkTok$=MID$(SpeicherZk$,BegPos%,EndPos%-BegPos%)
   ' Setze Startpunkt für die Suche nach dem
   ' nächsten Token.
   Start%=EndPos%
END FUNCTION
```

Ausgabe

```
Zeichenkette eingeben: Anton sprach: "Was ist eigentlich ein
Turbo?"
Anton
sprach
Was
ist
eigentlich
ein
Turbo
```

2.7 Typumwandlung

Bei Bedarf wandelt BASIC eine numerische Konstante nach folgenden Regeln von einem Typ in einen anderen um:

- Wenn eine numerische Konstante einer numerischen Variable eines anderen Typs übergeben wird, so wird die numerische Konstante in dem Typ gespeichert, mit dem die Variable deklariert wurde. Siehe hierzu folgendes Beispiel:

  ```
  A% = 23.43
  PRINT A%
  ```

  Ausgabe

  ```
  23
  ```

  Wird eine Zeichenkettenvariable jedoch einem numerischen Wert gleichgesetzt oder umgekehrt, so erscheint die Fehlermeldung `Unverträgliche Datentypen`.

- Während der Auswertung eines Ausdrucks werden alle Operanden in einer arithmetischen Operation oder einer Vergleichsoperation in denselben Genauigkeitsgrad, d. h. den Grad des genauesten Operanden, umgewandelt. Außerdem wird das Ergebnis einer arithmetischen Operation mit diesem Genauigkeitsgrad wiedergegeben, wie es im nachstehenden Beispiel zu sehen ist:

  ```
  X% = 2 : Y! = 1.5 : Z# = 100
  PRINT X% / Y! * Z#
  ```

Ausgabe

```
133.3333373069763
```

Obwohl das vorhergehende Ergebnis mit doppelter Genauigkeit (wegen der doppelten Genauigkeit der Variablen Z#) angezeigt wird, ist es nur von einfacher Genauigkeit, weil die erste Operation (X% / Y!) mit einfacher Genauigkeit berechnet wird. Dies erklärt die bedeutungslosen Ziffern (73069763) nach der fünften Dezimalstelle. Stellen Sie diesem Ergebnis die Ausgabe aus dem folgenden Beispiel gegenüber:

```
X% = 2 : Y# = 1.5 : Z# = 100
PRINT X% / Y# * Z#
```

Ausgabe

```
133.3333333333333
```

- Logische Operatoren wie **AND** und **NOT** wandeln bei Bedarf ihre Operanden in lange Ganzzahlen um. Operanden müssen innerhalb des Bereichs -2.147.483.648 bis +2.147.483.647 liegen, sonst erscheint die Fehlermeldung `Überlauf`. Weitere Informationen zu logischen Operatoren finden Sie in Kapitel 3, "Ausdrücke und Operatoren".
- Wird ein Gleitkommawert in eine ganze Zahl umgewandelt, so werden die Stellen hinter dem Dezimalpunkt wie im folgenden Beispiel gerundet:

```
Total% = 55.88
PRINT Total%
```

Ausgabe

```
56
```



3 Ausdrücke und Operatoren

Dieses Kapitel erklärt Ihnen, wie Sie Ausdrücke kombinieren, verändern, vergleichen oder Informationen über sie erhalten, indem Sie die in BASIC verfügbaren Operatoren benutzen.

Immer wenn Sie eine Berechnung durchführen oder eine Zeichenkette manipulieren, verwenden Sie Ausdrücke und Operatoren. Dieses Kapitel beschreibt, wie Ausdrücke geformt werden, bespricht die Reihenfolge, in der BASIC Operatoren benutzt, und schließt mit der Beschreibung der fünf folgenden Operatoren:

- Arithmetische Operatoren, die benutzt werden, um Berechnungen auszuführen
- Vergleichsoperatoren, die benutzt werden, um Zeichenketten und numerische Werte zu vergleichen
- Logische Operatoren, die benutzt werden, um komplexe Bedingungen zu testen oder einzelne Bits zu manipulieren
- Funktionale Operatoren, die benutzt werden, um einfache Operatoren zu ergänzen
- Zeichenkettenoperatoren, die benutzt werden, um Zeichenketten zu verknüpfen und zu vergleichen.

3.1 Ausdrücke und Operatoren

Ein Ausdruck kann eine Zeichenkettenkonstante oder numerische Konstante, eine Variable oder ein einzelner Wert sein, den Sie durch die Kombination von Konstanten, Variablen und anderen Ausdrücken mit Operatoren erhalten. Operatoren führen mathematische oder logische Operationen mit Werten durch. Die von BASIC bereitgestellten Operatoren können in folgende fünf Kategorien eingeteilt werden:

- Arithmetische Operatoren
- Vergleichsoperatoren
- Logische Operatoren
- Funktionale Operatoren
- Zeichenkettenoperatoren

3.2 Rangfolge der Operationen

Die BASIC-Operatoren haben eine bestimmte Rangfolge: Wenn innerhalb derselben Programmanweisung mehrere Operationen vorkommen, werden einige Operationen vor anderen ausgeführt. Hierzu gilt die nachstehende Reihenfolge.

- Arithmetische Operationen
  - a. Potenzierung (^)
  - b. Negation (-)
  - c. Multiplikation und Division (*,/)
  - d. Division von ganzen Zahlen (\)
  - e. Modulo-Arithmetik (**MOD**)
  - f. Addition und Subtraktion (+, -)
- Vergleichsoperationen (=, >, <, <>, <=, >=)
- Logische Operationen
  - a. **NOT**
  - b. **AND**
  - c. **OR**
  - d. **XOR**
  - e. **EQV**
  - f. **IMP**

Bei der vorstehend genannten Reihenfolge der Operationen gibt es eine Ausnahme: Wenn ein Ausdruck benachbarte Potenzierungs- und Negationsoperatoren hat. In diesem Fall wird die Negation zuerst ausgeführt. Die folgende Anweisung gibt beispielsweise den Wert 0,0625 (äquivalent zu 4^{-2}) und nicht -16 (äquivalent zu $-(4^{2})$) aus:

```
PRINT 4 ^ - 2
```

Wenn unterschiedliche Operationen auf gleicher Ebene erfolgen, wird die ganz links stehende Operation zuerst ausgeführt, die ganz rechts stehende zuletzt, wie im folgenden Beispiel:

```
A = 3 + 6 / 12 * 3 - 2    'A = 2.5
```

Die Reihenfolge der Operationen im vorhergehenden Beispiel lautet folgendermaßen:

1. 6 / 12 (= 0.5)
2. 0.5*3 (= 1.5)
3. 3 + 1.5 (= 4.5)
4. 4.5 - 2 (= 2.5)

Hinweis In einer Serie von Additionen oder Multiplikationen gibt es keine festgelegte Reihenfolge. In der folgenden Anweisung kann entweder 3 + 5 oder 5 + 6 zuerst berechnet werden:

```
C = 3 + 5 + 6
```

Üblicherweise verursacht dies keine Probleme. Es kann jedoch Probleme verursachen, wenn Sie eine Serie von **FUNCTION**-Prozeduraufrufen haben:

```
C = Inkr(X) + Dekr(X) + F(X)
```

Wenn eine der drei **FUNCTION**-Prozeduren X oder eine geteilte Variable verändert, hängt das Ergebnis von der Reihenfolge ab, in der BASIC Additionen ausführt. Sie können solche Situationen vermeiden, indem Sie die Ergebnisse der **FUNCTION**-Aufrufe temporären Variablen zuweisen und dann die Addition ausführen:

```
T1 = Inkr(X) : T2 = Dekr(X) : T3 = F(X)
C = T1 + T2 + T3
```

3.3 Arithmetische Operatoren

Klammern ändern die Reihenfolge, in der arithmetische Operationen ausgeführt werden. Operationen in Klammern werden zuerst ausgeführt. In Klammern wird die übliche Reihenfolge für Operationen beibehalten. Hier sind einige Beispiele von algebraischen Ausdrücken und ihren BASIC-Entsprechungen:

Algebraischer Ausdruck	*BASIC-Ausdruck*
$\frac{X-Y}{Z}$	(X-Y)/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X^2)^Y
X^{Y^Z}	X^(Y^Z)
X(-Y)	X*(-Y)

Im allgemeinen müssen zwei aufeinanderfolgende Operatoren durch Klammern getrennt werden. Die Ausnahmen zu dieser Regel sind * -, * +, ^ - und ^ +. Der letzte Ausdruck in der rechten Spalte oben könnte auch geschrieben werden als: `X*-Y`.

Informationen über die Reihenfolge, in der arithemtische Operationen ausgeführt werden, sind im vorhergehenden Kapitel enthalten.

3.3.1 Division von ganzen Zahlen

Die Division von ganzen Zahlen wird durch den rückwärtigen Schrägstrich (\) gekennzeichnet, nicht durch den normalen Schrägstrich (/), der eine Gleitkommadivision anzeigt. Bevor diese Division ausgeführt wird, werden die Operanden auf Ganzzahlen oder lange Ganzzahlen gerundet und müssen daher größer als -2.147.483.648,5 und kleiner als +2.147.483.647,5 sein. Der Quotient einer Ganzzahldivision wird auf eine ganze Zahl abgeschnitten.

Beispiel

```
PRINT 10\4, 10/4, -32768.499\10, -32768.499/10
```

Ausgabe

```
2       2.5      -3276      -3276.8499
```

3.3.2 Modulo-Arithmetik

Modulo-Arithmetik wird durch den Operator **MOD** gekennzeichnet. Modulo-Arithmetik gibt den Rest, nicht den Quotienten, einer Ganzzahldivision an.

Beispiel

```
X% = 10.4\4
REST% = INT(10.4) - 4*X%  '10\4 = 2, mit Rest 2
PRINT REST%, 10.4 MOD 4
```

Ausgabe

```
 2          2
```

3.3.3 Überlauf und Division durch Null

Division durch Null, Null mit einer negativen Zahl potenziert und arithmetischer Überlauf führen zu einem Laufzeitfehler. Diese Fehler können mit Hilfe einer Fehlerverfolgungsroutine erkannt werden. In Kapitel 6, "Fehler- und Ereignisverfolgung", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie weitere Informationen zum Schreiben von Fehlerverfolgungsroutinen.

3.4 Vergleichsoperatoren

Vergleichsoperatoren werden verwendet, um zwei Werte miteinander, wie in der folgenden Tabelle gezeigt, zu vergleichen. Das Ergebnis des Vergleichs ist entweder "wahr" (nicht Null) oder "falsch" (Null). Mit diesem Ergebnis kann dann eine Entscheidung hinsichtlich des Programmablaufs getroffen werden. Obwohl BASIC jeden Wert ungleich Null als wahr betrachtet, wird wahr üblicherweise durch -1 repräsentiert.

Operator	*Geprüfter Vergleich*	*Ausdruck*
=	Gleichheit*	X = Y
<>	Ungleichheit	X <> Y
<	Kleiner als	X < Y
>	Größer als	X > Y
<=	Kleiner gleich	X <= Y
>=	Größer gleich	X >= Y

* Das Gleichheitszeichen wird auch verwendet, um einer Variablen einen Wert zuzuweisen.

Werden arithmetische und Vergleichsoperationen in einem Ausdruck kombiniert, wird die arithemtische Operation immer zuerst ausgeführt. Zum Beispiel ist der folgende Ausdruck wahr, wenn der Wert von `X + Y` kleiner ist als der Wert von `(T - 1)/Z`:

```
X + Y < (T - 1) /Z
```

Seien Sie vorsichtig bei der Verwendung von Vergleichsoperatoren mit Werten einfacher und doppelter Genauigkeit. Berechnungen könnten nahezu gleiche aber nicht identische Ergebnisse liefern. Vermeiden Sie im besonderen den Vergleich auf Identität zwischen zwei Werten. Beispielsweise wird die **PRINT**-Anweisung in der folgenden **IF**-Anweisung nicht ausgeführt, solange `A!` nicht exakt 0.0 ist:

```
IF A! = 0.0 THEN PRINT "Genaues Ergebnis."
```

Wenn `A!` ein extrem kleiner Wert ist, beispielsweise 1,0E-23, wird die **PRINT**-Anweisung nicht ausgeführt.

Außerdem kann ein Programm als **.EXE**-Datei andere Ergebnisse liefern als bei Ausführung in der QuickBASIC-Umgebung. QuickBASIC liefert in einer **.EXE**-Datei effizienteren Code, der die Art, wie Werte einfacher und doppelter Genauigkeit verglichen werden, verändern kann. Beispielsweise schreibt der folgende Programmausschnitt `Gleich`, wenn er in der Umgebung läuft, jedoch `Ungleich`, wenn er als **.EXE**-Datei läuft:

```
B!=1.0
A!=B!/3.0
.
.
.
IF A!=B!/3.0 THEN PRINT "Gleich" ELSE PRINT "Ungleich"
```

Da die **.EXE**-Version intensiveren Gebrauch von einem Koprozessor bzw. dessen Emulation macht, haben `A!` und das Ergebnis von `B!/3.0` leicht unterschiedliche Werte.

Sie können dieses Problem vermeiden, indem Sie die Berechnungen außerhalb des Vergleichs vornehmen. Der folgende Programmausschnitt führt zu denselben Ergebnissen in der Umgebung und als **.EXE**-Datei:

```
B!=1.0
A!=B!/3.0
.
.
.
Tmp!=B!/3.0
IF A!=Tmp! THEN PRINT "Gleich" ELSE PRINT "Ungleich"
```

3.5 Logische Operatoren

Logische Operatoren führen Prüfungen auf Mehrfachrelationen, Bit-Manipulationen oder Boolesche Operationen durch und liefern einen "wahren" (nicht Null) oder "falschen" (Null) Wert zurück, der bei der Entscheidungsfindung benutzt werden kann.

Beispiele

```
IF D < 200 AND F < 4 THEN 80
WHILE I > 10 OR K < 0
.
.
.
WEND
IF NOT P THEN PRINT "Name nicht gefunden"
```

In BASIC gibt es sechs logische Operatoren, die nachstehend in ihrer Rangfolge aufgelistet werden:

Operator	*Bedeutung*
NOT	Logisches Komplement
AND	Konjunktion ("Und")
OR	Disjunktion (inklusives "Oder")
XOR	Exklusives "Oder"
EQV	Äquivalenz
IMP	Implikation

Jeder Operator liefert Ergebnisse zurück, die in der folgenden Tabelle angegeben sind. Ein "W" bezeichnet einen "wahren" und ein "F" einen "falschen" Wert. Die Operatoren sind in der Operatorenrangfolge aufgelistet.

X	*Y*	*NOT X*	*X AND Y*	*X OR Y*	*X XOR Y*	*X EQV Y*	*X IMP Y*
W	W	F	W	W	F	W	W
W	F	F	F	W	W	F	F
F	W	W	F	W	W	F	W
F	F	W	F	F	F	W	W

In einem Ausdruck werden logische Operationen (auch als Boolesche Operationen bezeichnet) nach arithmetischen Operationen und Vergleichen durchgeführt. Die Operanden von logischen Operatoren müssen im Bereich von -2.147.483.648 bis +2.147.483.647 liegen. Operanden werden in eine Ganzzahl (oder, wenn erforderlich, lange Ganzzahl) umgewandelt, bevor die Operation ausgeführt wird. (Bei Operanden außerhalb dieses Bereichs erscheint eine Fehlermeldung.) Wenn beide Operanden mit 0 oder -1 angegeben werden, ergeben logische Operatoren, wie im nachstehenden Beispiel, 0 oder -1. (Beachten Sie die Ähnlichkeit der Ausgabe dieses Programms mit der obigen Tabelle, "W" wird -1 und "F" wird 0.)

Beispiel

```
PRINT " X        Y     NOT    AND     OR      ";
PRINT "XOR   EQV   IMP"
PRINT
I = 10 : J = 15
X = (I = 10) : Y = (J = 15)    'X ist wahr (-1);
                               'Y ist wahr (-1)
CALL WahrhTafel (X,Y)
X = (I > 9) : Y = (J > 15)     'X ist wahr (-1);
                               'Y ist falsch (0)
CALL WahrhTafel (X,Y)
X = (I <> 10) : Y = (J < 16)     'X ist falsch (0);
                                 'Y ist wahr (-1)
CALL WahrhTafel (X,Y)
X = (I < 10) : Y = (J < 15)     'X ist falsch (0);
                                'Y ist falsch (0)
CALL WahrhTafel (X,Y)
END
SUB WahrhTafel(X,Y) STATIC
   PRINT X "      " Y "      ";NOT X "      " X AND Y;
   PRINT "      " X OR Y; "      " X XOR Y "     ";
   PRINT X EQV Y "    " X IMP Y
   PRINT
END SUB
```

Ausgabe

X	Y	NOT	AND	OR	XOR	EQV	IMP
-1	-1	0	-1	-1	0	-1	-1
-1	0	0	0	-1	-1	0	0
0	-1	-1	0	-1	-1	0	-1
0	0	-1	0	0	0	-1	-1

Logische Operatoren vergleichen jedes Bit des ersten Operanden mit dem entsprechenden Bit des zweiten Operanden, um das Bit im Ergebnis zu berechnen. In diesen bitweisen Vergleichen entspricht ein Bit 0 einem Wert "falsch" (F) in der Tabelle auf der vorhergehenden Seite, während ein Bit 1 einem Wert "wahr" (W) entspricht.

Es ist möglich, mit logischen Operatoren Bytes auf ein bestimmtes Bitmuster zu testen. Der Operator **AND** kann zum Beispiel dazu verwendet werden, alle Bits eines Statusbytes bis auf ein Bit zu maskieren, während der Operator **OR** dazu dienen kann, zwei Bytes zu einem bestimmten Binärwert zu verknüpfen.

Beispiel

```
PRINT 63 AND 16
PRINT -1 AND 8
PRINT 10 OR 9
PRINT 10 XOR 10              ' Immer 0
PRINT NOT 10, NOT 11, NOT 0  ' NOT X = -(x+1)
```

Ausgabe

```
16
8
11
0             -11            -12            -1
```

Die erste **PRINT**-Anweisung verwendet **AND**, um 63 (binär 111111) und 16 (10000) zu kombinieren. Wenn BASIC das Ergebnis eines **AND** berechnet, kombiniert es die Zahlen bitweise, wobei sich nur dann eine Eins ergibt, wenn beide Bits eins sind. Weil nur das fünfte Bit in beiden Zahlen eine Eins ist, ist im Ergebnis nur das fünfte Bit eine Eins. Das Ergebnis ist 16 oder 10000 binär. In der zweiten **PRINT**-Anweisung werden die Zahlen -1 (binär 1111111111111111) und 8 (binär 1000) kombiniert, indem eine andere **AND**-Operation ausgeführt wird. Das vierte Bit ist das einzige Bit, das in beiden Zahlen eins ist, daher ist das Ergebnis dezimal 8 oder binär 1000. Die dritte **PRINT**-Anweisung verwendet ein **OR**, um 10 (binär 1010) und 9 (binär 1001) zu verknüpfen. Ein **OR** ergibt ein Eins-Bit immer dann, wenn mindestens eines der Bits eins ist, das Ergebnis des **OR** im dritten **PRINT** ist daher 11 (binär 1011). Das **XOR** im vierten **PRINT** verknüpft die Zahl 10 (binär 1010) mit sich selbst. Das Ergebnis ist eine Null, weil ein **XOR** nur dann eine Eins ergibt, wenn genau eines der Bits 1 ist.

Wird mit einer Zahl ein **NOT** ausgeführt, ändern sich alle Eins-Bits zu Null und alle Null-Bits zu Eins. Wegen der Art und Weise, wie ein Zweierkomplement arbeitet, ist das Ausführen eines **NOT** für einen Wert das gleiche wie die Addition von 1 zu der Zahl mit anschließender Negation des Ergebnisses. Daher ergibt in der letzten **PRINT**-Anweisung der Ausdruck `NOT 10` ein Ergebnis von -11.

3.6 Funktionale Operatoren

Eine Funktion wird in einem Ausdruck verwendet, um eine vorher bestimmte Operation aufzurufen, die in einem Operanden ausgeführt werden soll. Beispielsweise ist **SQR** ein funktionaler Operator, der in der nachstehenden Zuweisungsanweisung zweimal benutzt wird:

```
A = SQR (20.25) + SQR (37)
```

BASIC enthält zwei Funktionsarten: interne Standardfunktionen und vom Benutzer definierte Funktionen. BASIC hat viele vordefinierte Funktionen in die Sprache eingebaut. Beispiele hierfür sind die Funktionen **SQR** (Quadratwurzel) und **SIN** (Sinus).

Sie können Ihre eigenen Funktionen mit Hilfe der **FUNCTION...END FUNCTION** und der älteren, überholten **DEF FN...END DEF**-Konstruktion definieren. Solche Funktionen sind nur für die Dauer des Programms definiert (soweit sie nicht in einer Quick-Bibliothek sind) und gehören nicht zur BASIC-Sprache. Außer **FUNCTION** und **DEF FN** ermöglicht BASIC auch die Definition von Unterprogrammen mit **SUB**. Weitere Informationen über die Definition Ihrer eigenen Funktionen und Unterprogramme finden Sie in diesem Handbuch in dem Teil 2, "Nachschlageteil – Anweisungen und Funktionen", und Kapitel 4, "Programme und Module", sowie in Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen.*

3.7 Zeichenkettenoperatoren

Ein Zeichenkettenausdruck besteht aus Zeichenkettenkonstanten, Zeichenkettenvariablen und anderen Zeichenkettenausdrücken, die mit Zeichenkettenoperatoren verbunden sind. Es gibt zwei Arten von Zeichenkettenoperationen: Verkettung und Zeichenkettenfunktionen.

Die Kombination zweier Zeichenketten nennt man Verkettung. Das Pluszeichen (+) ist der Verkettungsoperator für Zeichenketten. Das folgende Programmfragment verbindet die Zeichenkettenvariablen `A$` und `B$` zu dem Wert `DATEINAME`.

```
A$ = "DATEI": B$ = "NAME"
PRINT A$ + B$
PRINT "NEUER " + A$ + B$
```

Ausgabe

```
DATEINAME
NEUER DATEINAME
```

Zeichenketten können mit folgenden Vergleichsoperatoren verglichen werden (siehe Tabelle in Abschnitt 3.4, "Vergleichsoperatoren"):

<> = < > <= >=

Beachten Sie, daß dies die gleichen Vergleichsoperatoren sind, die auch bei Zahlen verwendet werden.

Zeichenketten werden miteinander verglichen, indem man die entsprechenden Zeichen aus jeder Folge herausnimmt und ihre ASCII-Codes miteinander vergleicht. Wenn die ASCII-Codes für alle Zeichen in beiden Zeichenketten gleich sind, sind die Zeichenketten gleich. Sind die ASCII-Codes verschieden, so hat die niedrigere Codenummer Vorrang vor der höheren. Wenn während eines Zeichenkettenvergleichs das Ende einer Zeichenkette erreicht wird, gilt die kürzere Zeichenkette als die kleinere, wenn die beiden Folgen bis zu dieser Stelle gleich sind. Führende und nachfolgende Leerzeichen sind wichtig. Die folgenden Beispiele zeigen "wahre" Ausdrücke:

```
"AA" < "AB"
"DATEINAME" = "DATEI" + "NAME"
"X&" > "X#"
"CL " > "CL"
"kg" > "KG"
"SCHMID" < "SCHMIDT"
B$ < "9/12/78"       'Wobei B$ = "8/12/85"
```

Auf diese Weise können Zeichenkettenvergleiche benutzt werden, um Zeichenkettenwerte zu testen oder Zeichenketten zu alphabetisieren. Alle in Vergleichsausdrücken verwendeten Konstanten müssen in Anführungszeichen stehen. Weitere Informationen zum ASCII-Code finden Sie im Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes".

4 Programme und Module

Dieses Kapitel beschreibt die größten Teile eines Programms - Module und Prozeduren. Hier wird gezeigt, wie Module organisiert sind und wie BASIC-Prozeduren mit anderen Teilen des Programms kommunizieren.

Dieses Kapitel erklärt im einzelnen die folgenden Themen:

- Organisation der Module
- Typen von BASIC-Prozeduren
- Übergabe von Argumenten an eine Prozedur
- Rekursion

4.1 Module

BASIC-Programme bestehen aus einem oder mehreren Modulen. Ein Modul ist eine Quelldatei, die separat kompiliert werden kann. Deklarationen, ausführbare Anweisungen – jede BASIC-Anweisung – können in einem Modul vorkommen.

Ein Modul kann **SUB**- und **FUNCTION**-Prozeduren enthalten und Code, der nicht direkt Teil einer **SUB** oder **FUNCTION** ist. Anweisungen, die nicht Teil einer **SUB** oder **FUNCTION** sind, werden als Modul-Ebenen-Code bezeichnet. Modul-Ebenen-Code schließt sowohl deklarierende Anweisungen wie **DIM** und **TYPE** als auch Fehlerbehandlungs- und Ereignisbehandlungscode ein.

Ein Programm hat ein besonderes Modul – das Hauptmodul. Das Hauptmodul enthält den Eingangspunkt des Programms (der Punkt, an dem der Programmablauf beginnt). Jedes Programm enthält nur ein Hauptmodul. Im Hauptmodul entspricht Modul-Ebenen-Code dem, was häufig als Hauptprogramm bezeichnet wird.

4.2 Prozeduren

Dieser Abschnitt beschreibt die älteren **DEF FN**-Funktionen und vergleicht diese mit den beiden neueren BASIC-Prozeduren: **FUNCTION**-Prozeduren und **SUB**-Prozeduren.

In Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie detaillierte Beispiele und Informationen zu dem jeweiligen Gebrauch verschiedener Prozeduren.

4.2.1 DEF FN-Funktionen

DEF FN-Funktionen sind immer Teil des Modul-Ebenen-Codes. Wie **FUNCTION**-Prozeduren geben **DEF FN**-Funktionen einen Wert zurück und sind wie eingebaute BASIC-Funktionen zu verwenden:

```
' Funktion, um von einer Zahl den Logarithmus zur
' Basis 10 unter Benutzung der eingebauten BASIC-
' Funktion Natürlicher Logarithmus zu berechnen.
DEF FNLog10 (X)
   FNLog10=LOG(X)/LOG(10.0)
END DEF
INPUT "Bitte eine Zahl eingeben: ",Num
PRINT "10 ^ Log10(";Num;") ist" 10.0^FNLog10(Num)
END
```

DEF FN-Funktionsargumente werden als Wert übergeben. (Siehe Abschnitt 4.3, "Referenzübergabe und Wertübergabe", für weitere Informationen.) Der Name einer **DEF FN**-Funktion beginnt immer mit **FN. DEF FN**-Funktionen können nicht rekursiv verwendet werden und müssen definiert sein, bevor sie benutzt werden können. **DEF FN**-Funktionen können nur innerhalb des Moduls aufgerufen werden, in dem sie definiert wurden.

4.2.2 FUNCTION-Prozeduren

FUNCTION-Prozeduren bieten eine hervorragende Alternative zu **DEF FN**-Funktionen. **FUNCTION**-Prozeduren wie auch **DEF FN**-Funktionen werden in Ausdrücken benutzt und geben direkt einen einzigen Wert zurück. Es gibt jedoch wichtige Unterschiede.

FUNCTION-Prozeduren verwenden die Übergabe als Referenz. Somit kann eine **FUNCTION** zusätzliche Werte zurückgeben, indem sie Variablen in ihrer Argumentenliste ändert. Zusätzlich können **FUNCTION**-Prozeduren rekursiv benutzt werden - eine Funktion kann sich selbst aufrufen (siehe 4.4, "Rekursion").

Anders als **DEF FN**-Funktionen kann eine **FUNCTION**-Prozedur außerhalb des Moduls aufgerufen werden, in dem sie definiert wurde. Sie müssen jedoch noch eine **DECLARE**-Anweisung aufnehmen, wenn Sie eine **FUNCTION** benutzen, die in einem anderen Modul definiert wurde. QuickBASIC erzeugt automatisch **DECLARE**-Anweisungen für **FUNCTION**-Prozeduren, die im selben Modul definert und benutzt werden. Sie können **DECLARE** jedoch auch selbst eintippen:

```
DECLARE FUNCTION Log10(X)
INPUT "Geben Sie eine Zahl ein: ",Num
PRINT "10 ^ Log10(";Num;") ist" 10.0^Log10(Num)
END
' Funktion, um von einer Zahl den Logarithmus zur
' Basis 10 unter Benutzung der eingebauten BASIC-
' Funktion Natürlicher Logarithmus zu berechnen.
FUNCTION LOG10 (X) STATIC
   Log10=LOG(X)/LOG(10.0)
END FUNCTION
```

FUNCTION-Prozeduren unterscheiden sich von **DEF FN**-Funktionen noch in der Weise, daß sie nicht Teil des Modul-Ebenen-Codes sind.

4.2.3 SUB-Prozeduren

Im Unterschied zu **DEF FN**- und **FUNCTION**-Prozeduren wird **SUB** als eine eigenständige Anweisung aufgerufen:

```
' Gib eine Meldung in der Mitte des Bildschirms aus.
CLS
CALL AusgMeldung(12,40,"Hallo!")
END
' Gib eine Meldung in der dafür vorgesehenen Zeile und
' Spalte aus.
SUB AusgMeldung(Zeile%,Spalte%,Meldung$) STATIC
   ' Sichern der aktuellen Cursor-Position
   CurZeile%=CSRLIN
   CurSpalte%=POS(0)
   ' Schreibe die Meldung an die Position
   LOCATE Zeile%,Spalte% : Print Meldung$;
   ' Plaziere Cursor auf alter Position
   LOCATE CurZeile%,CurSpalte%
END SUB
```

SUB-Prozeduren können zur Rückgabe mehrerer Werte zu der aufrufenden Routine benutzt werden; sie dürfen jedoch nicht Teil eines Ausdrucks sein.

Alle **SUB**-Argumente werden als Referenz übergeben. Dies erlaubt **SUB**-Prozeduren, Werte zurückzugeben, indem Variablen der Argumentenliste verändert werden. Dies ist die einzige Möglichkeit, mit der **SUB** einen Wert zurückgeben kann.

Sie können **SUB**-Prozeduren mit einer **CALL**-Anweisung ohne das **CALL**-Schlüsselwort aufrufen, wenn **SUB** deklariert ist.

```
DECLARE SUB AusgMeldung (Zeile%,Spalte%,Meldung%)
' Gib eine Meldung in der Mitte des Bildschirms aus.
CLS
AusgMeldung 12,40,"Hallo!"
END
.
.
.
```

SUB-Prozeduren können rekursiv benutzt werden – Sie können eine **SUB**-Prozedur schreiben, die sich selbst aufruft.

4.3 Referenzübergabe und Wertübergabe

BASIC benutzt zwei verschiedene Arten, um Argumente an Prozeduren zu übergeben. Der Ausdruck "Referenzübergabe" wird von **SUB**- und **FUNCTION**-Prozeduren benutzt. Darunter versteht man, daß die *Adresse* eines jeden Arguments, welches an die Prozedur übergeben wird, als Adresse auf den Stapel gelegt wird. Der bei **DEF FN**-Funktionen benutzte Ausdruck "Wertübergabe" zeigt an, daß der *Wert* und nicht die Adresse des Arguments auf den Stapel gelegt wird. Da die Prozedur keinen Zugriff auf die Variable hat, wenn ein Argument als Wert übergeben wird, kann die Prozedur den Wert der Variablen nicht verändern.

Manchmal ist es bequem, ein Argument als Wert an eine **FUNCTION** oder **SUB** zu übergeben. Sie können die Übergabe als Wert simulieren, indem Sie einen Ausdruck in dem **SUB**-Aufruf oder im **FUNCTION**-Aufruf benutzen:

```
Xkoordinate=Transform ((A#))
```

Da (`A#`) ein Ausdruck ist, errechnet BASIC den Wert `A#` und übergibt die Adresse des vorübergehenden Bereiches, der den Wert enhält. Eine Adresse wird dennoch übergeben. Da es aber die Adresse eines vorübergehenden Bereiches ist, wird die Übergabe als Wert simuliert.

Im Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie detaillierte Informationen, wie sich die BASIC-Aufrufvereinbarungen von denen anderer Sprachen unterscheiden.

4.4 Rekursion

BASIC erlaubt es Ihnen, rekursive **SUB**- und **FUNCTION**-Prozeduren (eine Prozedur, die sich selbst aufruft) zu schreiben. Eine rekursive Prozedur benutzt eine Strategie des "divide and conquer": Kompliziertere Probleme werden in einfachere Probleme aufgeteilt, die dann gelöst werden.

Das folgende Programm beispielsweise benutzt eine rekursive **FUNCTION**, um eine Zeichenkette in umgekehrter Reihenfolge darzustellen.

```
DECLARE FUNCTION Umkehrung$ (StringVar$)
PRINT "Gib die umzukehrende Zeichenkette ein ";
LINE INPUT ":",X$
PRINT Umkehrung$(X$)
END
FUNCTION Umkehrung$ (S$)
   C$ = MID$(S$, 1, 1)
   IF C$ = "" THEN
      ' Das erste Zeichen ist leer, gib leer zurueck-
      ' es gibt keine Zeichen mehr.
      Umkehrung$ =""
   ELSE
      ' Die Umkehrung einer nicht leeren Zeichenkette
      ' ist das erste Zeichen angehaengt an die
      ' Umkehrung der verbleibenden Zeichenkette.
      Umkehrung$ = Umkehrung$(MID$(S$, 2)) + C$
   END IF
END FUNCTION
```

Ausgabe

```
Gib die umzukehrende Zeichenkette ein:
abcdefgh...tuvwyxz

zyxwvut...hgfedcba
```

`Umkehrung$` kehrt eine Zeichenkette um, indem zunächst der einfachste Fall überprüft wird - eine Nullzeichenkette. Wenn die Zeichenkette Null ist, wird eine Null-Zeichenkette ausgegeben. Wenn die Zeichenkette nicht Null ist, also Zeichen enthält, dann vereinfacht `Umkehrung$` das Problem. Die Umkehrung einer Nicht-Null-Zeichenkette ist das erste Zeichen der Zeichenkette (`C$`), verbunden mit der Umkehrung der restlichen Zeichenkette. Also ruft `Umkehrung$` sich selbst auf, um den Rest der Zeichenkette umzukehren, und wenn dies geschehen ist, wird das erste Zeichen mit der umgekehrten Zeichenkette verbunden.

Rekursion kann viel Speicherplatz in Anspruch nehmen, da automatische Variablen innerhalb der **FUNCTION** oder **SUB** gespeichert werden müssen, damit die Prozedur erneut gestartet werden kann, wenn der rekursive Aufruf beendet ist. Da automatische Variablen auf dem Stapel gespeichert werden, müssen Sie eventuell die Stapelgröße mit der **CLEAR**-Anweisung heraufsetzen, um stets genügend Stapelplatz zu haben. Mit der **FRE**-Funktion können Sie bestimmen, um wieviel Sie die Stapelgröße anpassen müssen.



Teil 2: Nachschlageteil — Anweisungen und Funktionen

Teil 2 ist ein Nachschlagewerk der Microsoft BASIC-Anweisungen und -Funktionen. Von **ABS** bis **WRITE** werden Wirkungen, Syntax und Verwendung beschrieben. Außerdem finden Sie hier hilfreiche Beispiele. Auf den Nachschlageseiten werden neuere und effizientere Gegenstücke zu BASICA-Anweisungen erwähnt und entsprechende Querverweise gegeben.

ABS-Funktion

Funktion

Gibt den absoluten Wert eines numerischen Ausdrucks an.

Syntax

ABS(*Numerischer Ausdruck*)

Anmerkungen

Die Absolutwert-Funktion gibt die vorzeichenlose Größe ihres Arguments an. So sind zum Beispiel `ABS(-1)` und `ABS(1)` beide 1.

Beispiel

Das folgende Beispiel berechnet den näherungsweisen Wert einer Quadratwurzel. Es benutzt **ABS**, um die Differenz zwischen zwei Näherungen zu berechnen und stellt anhand der Differenz fest, ob die aktuelle Näherung genau genug ist.

```
DEFDBL A-Z
FUNCTION WurzelDrei(Wert,Genauigkeit) STATIC
  'Weise die ersten beiden Werte zu.
   X1=0.0# : X2=Wert
  'Wiederhole, bis die Differenz zwischen zwei Werten
  'kleiner als die geforderte Genauigkeit ist.
   DO UNTIL ABS(X1-X2) < Genauigkeit
      X=(X1+X2)/2.0#
        'Passe die Werte an.
      IF X*X*X-Wert < 0.0# THEN
         X1=X
      ELSE
         X2=X
      END IF
   LOOP
   WurzelDrei=X
END FUNCTION
INPUT "Gib einen Wert ein: ",X
PRINT "Die dritte Wurzel ist ";WurzelDrei(X,.0000001#)
```

Ausgabe

```
Gib einen Wert ein: 27
Die dritte Wurzel ist 2.999999972060323
```

ASC-Funktion

Funktion

Gibt einen numerischen Wert an, welcher der ASCII-Code für das erste Zeichen der Zeichenkette ist.

Syntax

ASC (*Zeichenkettenausdruck*)

Anmerkungen

ASC gibt eine Laufzeitfehlermeldung aus (`Unzulässiger Funktionsaufruf`), wenn *Zeichenkettenausdruck* gleich Null ist.

Vergleichen Sie auch

CHR$; Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes"

Beispiel

Das folgende Beispiel benutzt **ASC**, um den Hash-Wert, ein Indexwert zu einer Tabelle oder Datei, zu berechnen:

```
CONST HASHTABGROSS=101
FUNCTION HashWert(S$,Groesse) STATIC
   TempWert=0
   FOR I=1 TO LEN(S$)
         'Konvertiere die Zeichenkette in eine Zahl
         'durch Summierung der Werte der einzelnen
         'Buchstaben
      TempWert=TempWert+ASC(MID$(S$,I,1))
   NEXT I
```

```
    'Dividiere die Summe durch die Größe der Tabelle
    HashWert=TempWert MOD Groesse
END FUNCTION
INPUT "Gib einen Namen ein: ",Nm$
PRINT "Der Hash-Wert ist ";HashWert(Nm$,HASHTABGROSS)
```

Ausgabe

```
Gib einen Namen ein: Bafflegab
Der Hash-Wert ist  66
```

ATN-Funktion

Funktion

Gibt den Arkustangens von Numerischer Ausdruck an, (der Winkel, dessen Tangens Numerischer Ausdruck entspricht).

Syntax

ATN (*Numerischer Ausdruck*)

Anmerkungen

Numerischer Ausdruck kann von jedem numerischen Typ sein.

Das Ergebnis wird im Bogenmaß angegeben und liegt im Bereich von - $\pi/2$ bis $\pi/2$ rad, wobei π = 3,141593 ist. **ATN** wird standardmäßig (default) mit einfacher Genauigkeit ausgewertet. Wenn *Numerischer Ausdruck* ein Wert doppelter Genauigkeit ist, wird **ATN** mit doppelter Genauigkeit ausgewertet.

Beispiel

Das folgende Beispiel berechnet zuerst den Tangens von $\pi/4$ und anschließend den Arkustangens dieses Wertes. Das Ergebnis ist $\pi/4$.

```
CONST PI=3.141592653
PRINT ATN(TAN(PI/4.0)), PI/4.0
```

Ausgabe

```
.78539816325  .78539816325
```

BEEP-Anweisung

Funktion

Sendet ein Tonsignal an den Lautsprecher.

Syntax

BEEP

Anmerkungen

Die **BEEP**-Anweisung erzeugt einen Ton im Lautsprecher. Diese Anweisung hat dieselbe Wirkung wie die folgende:

```
PRINT CHR$(7)
```

Beispiel

Das folgende Beispiel benutzt **BEEP**, um einen Eingabefehler anzuzeigen.

```
DO
   INPUT "Weiter (J oder N)";Antwort$
   A$=UCASE$(MID$(Antwort$,1,1))
   IF A$="J" OR A$="N" THEN EXIT DO
   BEEP
LOOP
```

BLOAD-Anweisung

Funktion

Lädt eine Speicherabbilddatei, die mit **BSAVE** erstellt wurde, von einer Eingabedatei oder einem Eingabegerät in den Speicher.

Syntax

BLOAD *Dateiangabe* [,*Offset*]

Anmerkungen

Die **BLOAD**-Anweisung hat folgende Argumente:

Argument	*Beschreibung*
Dateiangabe	Eine Zeichenkette, die die Dateibezeichnung enthält. Auch andere Eingabegeräte als die Tastatur (**KYBD:**) werden unterstützt.
Offset	Der Offset der Adresse, ab der die Datei geladen werden soll.

Mit der **BLOAD**-Anweisung können Programme oder Daten, die als Speicherabbilddatei gespeichert wurden, an jede beliebige Stelle in den Speicher geladen werden. Eine Speicherabbilddatei ist eine byte-weise Kopie des ursprünglichen Speicherinhalts.

Wichtig Programme, die in früheren BASIC-Versionen geschriebenen wurden, laufen nicht mehr, falls sie **VARPTR** für den Zugriff auf numerische Datenfelder benutzen.

Die Startadresse für das Laden wird durch den angegebenen Offset und der letzten **DEF SEG**-Anweisung bestimmt. Wenn der *Offset* nicht angegeben wird, werden die Segmentadresse und der Offset, die in der Datei enthalten sind, verwendet (d. h. die Adresse, die in der **BSAVE**-Anweisung angegeben wurde). Deshalb wird die Datei an dieselbe Adresse geladen, an der sie gespeichert wurde.

Wenn Sie einen Offset angeben, ist die benutzte Segmentadresse das mit der zuletzt ausgeführten **DEF SEG**-Anweisung angegebene Segment. Wenn keine **DEF SEG**-Anweisung gegeben wurde, wird das BASIC-Datensegment (**DS**) als Vorgabe benutzt.

Falls der Offset eine Zahl mit einfacher oder doppelter Genauigkeit ist, wird er in eine Ganzzahl umgewandelt. Falls der Offset eine negative Zahl im Bereich -1 bis -32768 ist, wird er als vorzeichenloser 2-Byte-Offset behandelt.

Wichtig Weil **BLOAD** keine Adreßbereichsprüfung durchführt, ist es möglich, daß eine Datei an eine beliebige Speicherstelle geladen wird. Sie müssen darauf achten, BASIC oder das Betriebssystem nicht zu überschreiben.

Weil verschiedene Bildschirm-Modi den Speicher unterschiedlich nutzen, sollten Sie grafische Darstellungen nicht in einem anderen Bildschirm-Modus als den bei der Erstellung benutzten laden.

Außerdem sollten Sie **BLOAD** nicht für Dateien benutzen, die von BASICA-Programmen angelegt wurden, weil BASIC-Programmcode und -Datenobjekte nicht immer an der gleichen Adresse wie in BASICA geladen werden.

Unterschiede zu BASICA

BLOAD unterstützt keine Kassettengeräte.

Vergleichen Sie auch

BSAVE, DEF SEG, VARPTR, VARSEG

Beispiel

Dieses Beispiel benutzt **BLOAD**, um eine mit **BSAVE** auf Diskette oder Festplatte gespeicherte Zeichnung wiederherzustellen. Vergleichen Sie auch das Beispiel für **BSAVE**, um zu sehen, wie die Zeichnung gespeichert wurde.

```
DIM Wuerfel(1 TO 675)
'Setze den Bildschirmmodus - der Modus sollte der
'gleiche sein, mit dem die ursprüngliche Zeichnung
'erstellt wurde.
SCREEN 1
'Lade die Zeichnung in das Datenfeld Würfel.
DEF SEG=VARSEG(Wuerfel(1)) 'Ermittle das Segment des
                           'Datenfeldes.
BLOAD "zwuerfel.gra", VARPTR(Wuerfel(1))
DEF SEG    'Stelle das Standard-Segment wieder her.
'Gib die Zeichnung auf dem Bildschirm aus.
PUT (80,10),Wuerfel
```

BSAVE-Anweisung

Funktion

Übergibt den Inhalt eines Speicherbereichs an eine Ausgabedatei oder ein Ausgabegerät.

Syntax

BSAVE *Dateiangabe, Offset, Länge*

Anmerkungen

BSAVE-Anweisungen haben folgende Argumente:

Argument	*Beschreibung*
Dateiangabe	Zeichenkettenausdruck, der den Datei- oder Gerätenamen enthält. Auch andere Ausgabegeräte als die Konsole (**SCRN:** und **CONS:**) werden unterstützt.
Offset	Der Offset der Startadresse des Speicherbereichs, der gesichert werden soll.
Länge	Die Anzahl der zu speichernden Bytes. Dies ist ein numerischer Ausdruck, der eine vorzeichenlose ganze Zahl im Bereich von 0 bis 65.535 angibt.

Mit der **BSAVE**-Anweisung kann man Daten oder Programme als Speicherabbilddatei auf Diskette oder Festplatte speichern. Eine Speicherabbilddatei ist eine byte-weise Kopie des Speicherinhalts, zusammen mit Steuer-Informationen, die **BLOAD** zum Laden der Datei benutzt.

Wichtig Programme, die in früheren BASIC-Versionen geschrieben wurden, laufen nicht mehr, falls sie **VARPTR** für den Zugriff auf numerische Datenfelder benutzen.

Die Startadresse des zu speichernden Bereichs wird durch den angegebenen Offset und der letzten **DEF SEG**-Anweisung bestimmt.

Wenn keine **DEF SEG**-Anweisung vor der **BSAVE**-Anweisung ausgeführt wurde, benutzt das Programm das vorgegebene BASIC-Datensegment (**DS**). Andernfalls beginnt **BSAVE** das Speichern an der Adresse, die von dem Offset und dem von der letzten **DEF SEG**-Anweisung festgelegten Segment bestimmt wird.

Falls der Offset eine Zahl mit einfacher oder doppelter Genauigkeit ist, wird er in eine Ganzzahl umgewandelt. Falls der Offset eine negative Zahl im Bereich -1 bis -32768 ist, wird er als vorzeichenloser 2-Byte-Offset behandelt.

Wichtig Weil verschiedene Bildschirm-Modi den Speicher unterschiedlich nutzen, sollten Sie grafische Darstellungen nicht in einem anderen Bildschirm-Modus als den bei der Erstellung benutzten laden.

Unterschiede zu BASICA

BSAVE unterstützt keine Kassettengeräte.

Vergleichen Sie auch

BLOAD

Beispiel

Dieses Beispiel zeichnet eine Grafik und sichert sie dann als Speicherabbilddatei. Wie diese Datei benutzt wird, finden Sie im Beispiel für die **BSAVE**-Anweisung.

```
'Dieses Programm zeichnet auf dem Bildschirm,
'speichert die Zeichnung in einem Datenfeld und
'verwendet BSAVE, um die Zeichnung dann in einer
'Plattendatei zu sichern.
DIM Wuerfel(1 TO 675)
SCREEN 1
'Zeichne einen weißen Kasten.
LINE (140,25)-(140+100,125),3,b
'Zeichne die Außenlinie eines violetten Würfels
'innerhalb des Kastens.
DRAW "C2 BM140,50 M+50,-25 M+50,25 M-50,25 M-50,_
      -25 M+0,50 M+50,25 M+50,-25 M+0,-50 BM190,75 M+0,50"
'Speichere die Zeichnung in dem Datenfeld Würfel.
GET (140,25)-(240,125),Wuerfel
'Sichere die Zeichnung in einer Plattendatei. Beachte:
'2700 ist die Anzahl der Bytes in Würfel (4 Bytes für
'jedes Datenfeldelement * 675).
DEF SEG=VARSEG(Wuerfel(1))  'Setze das Segment auf das
                            'Segment des Datenfeldes.
BSAVE "zwuerfel.gra",VARPTR(Wuerfel(1)),2700
DEF SEG   'Stelle BASIC-Segment wieder her.
```

CALL-Anweisung (BASIC-Prozeduren)

Funktion

Übergibt die Steuerung an eine BASIC-**SUB**.

Syntax 1

CALL *Name* [(*Argumentenliste*)]

Syntax 2

Name [(*Argumentenliste*)]

Anmerkungen

Die **CALL**-Anweisung hat folgende Argumente:

Argument	*Beschreibung*
Name	Der Name (weniger als 40 Zeichen) der aufzurufenden BASIC-**SUB**. Der Name muß in der **SUB**-Anweisung erscheinen, wenn die **SUB**-Anweisung im selben Modul ist.
Argumentenliste	Die Variablen oder Konstanten, die an das Unterprogramm übergeben werden. Argumente in der Liste werden durch Kommata getrennt. Als Referenz übergebene Argumente können im Unterprogramm geändert werden.

Wenn *Argumentenliste* ein Datenfeldargument enthält, wird das Datenfeld mit dem Datenfeldnamen, gefolgt von leeren Klammern, angegeben:

```
DIM ZahlDatenfeld(1 TO 20)
.
.
.
CALL ShellSort(ZahlDatenfeld())
```

Wenn Sie die **CALL**-Anweisung benutzen, ist das **CALL**-Schlüsselwort wahlfrei. Wenn Sie jedoch das **CALL**-Schlüsselwort weglassen, müssen Sie die Prozedur in einer **DECLARE**-Anweisung deklarieren. Beachten Sie auch, daß die Klammern um die Argumentenliste weggelassen werden müssen, falls Sie das **CALL**-Schlüsselwort weglassen. In Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie weitere Informationen.

Argumente werden als Referenz übergeben: Dem Unterprogramm wird die Adresse des Argumentes übergeben. Dies erlaubt Unterprogrammen, die Argumentwerte zu ändern. BASIC kann ebenso Argumente als Wert übergeben. Die folgende Anweisung ruft ein Unterprogramm auf und übergibt ein Argument als Wert :

```
CALL LoesePuzzle ((StartWert))
```

Weil `StartWert` in Klammern steht, wertet BASIC es als Ausdruck aus. Das Ergebnis wird in einem temporären Bereich gespeichert und die Adresse des temporären Bereichs wird an die **SUB** übergeben. Jede Änderung, die in `LoesePuzzle` vorgenommen wird, wird nur im temporären Bereich ausgeführt, der Variablenwert ändert sich nicht.

Vergleichen Sie auch

CALL, CALLS (Nicht-BASIC); CALL ABSOLUTE; DECLARE (BASIC)

Beispiel

Das folgende Programm kopiert eine Serie von Dateien in eine neue Datei. Die neue Datei bekommt den letzten Namen von der Befehlszeile. In dem Programm wird, nach dem Aufteilen der Befehlszeile in verschiedene Dateinamen und deren Speicherung in dem Datenfeld `Datei$`, das BASIC-Unterprogramm `Ausdruck` aufgerufen. Dieses kopiert die Dateien in die letzte Datei der Liste und zu dem Standardausgabegerät (defaultmäßig Ihr Bildschirm).

```
DEFINT A-Z
CONST MAXDATEI=5, FELDDIM=MAXDATEI+1
DIM Datei$(1 TO FELDDIM)
'Teile Befehlszeile in Argumente auf.
CALL BefZeile(AnzArg,Datei$(),FELDDIM)
'Prüfung auf zu viele oder zu wenige Dateien.
IF AnzArg < 3 OR AnzArg > MAXDATEI THEN
     'Zu viele oder zu wenige Dateien.
     PRINT "Benutze mehr als 3 und weniger als";
     PRINT MAXDATEI;"Dateien"
ELSE
   'Sende alle Dateien zu Ausdruck.
   CALL Ausdruck (Datei$(),AnzArg)
END IF
END
'Zur Definition von Befzeile siehe Beispiel unter
'COMMAND$. BefZeile würde hier erscheinen.
SUB Ausdruck(F$(1),N) STATIC
   'Öffne Zieldatei.
   OPEN F$(N) FOR OUTPUT AS #3
   'Schleife wird für jede Datei einmal ausgeführt.
   'Kopiere die ersten N-1 Dateien in die N-te Datei.
   FOR Datei = 1 TO N-1
      OPEN F$(Datei) FOR INPUT AS #1
      DO WHILE NOT EOF(1)
         'Lies Datei.
         LINE INPUT #1, Temp$
         'Schreibe Daten in Zieldatei.
         PRINT #3, Temp$
         PRINT Temp$          'Schreibe Datei in
                              'Standard-Ausgabe.
      LOOP
      CLOSE #1
   NEXT
   CLOSE
END SUB
```

CALL-, CALLS-Anweisungen (Nicht-BASIC-Prozeduren)

Funktion

Übergibt die Steuerung an eine Prozedur, die in einer anderen Sprache geschrieben ist.

Syntax 1

CALL *Name* [(*Aufruf-Argumentenliste*)]

Syntax 2

Name [*Aufruf-Argumentenliste*]

Syntax 3

CALLS *Name* [(*Aufrufe-Argumentenliste*)]

Anmerkungen

Die folgende Aufstellung beschreibt die Teile der **CALL**-Anweisung:

Argument	*Beschreibung*
Name	Der Name der aufzurufenden Prozedur. Ein Name darf höchstens 40 Zeichen lang sein.
Aufruf-Argumentenliste	Die Variablen oder Konstanten, die an die Prozedur übergeben werden. Die Syntax der *Aufruf-Argumentenliste* wird nachfolgend beschrieben.
Aufrufe-Argumentenliste	Eine durch Kommata getrennte Liste, die die Variablen und Konstanten enthält, die **CALLS** an die Prozedur übergibt. Beachten Sie bitte, daß diese Argumente als Referenz (lange Adressen) übergeben werden – unter Benutzung des Segments und des Offsets der Variablen. Sie können **BYVAL** oder **SEG** in der *Aufrufe-Argumentenliste* nicht verwenden.

Eine *Aufruf-Argumentenliste* hat die folgende Syntax:

[[{**BYVAL** | **SEG**}] *Argument*][, [{**BYVAL** | **SEG**}],*Argument*]...

Die folgende Liste beschreibt die Teile der *Aufruf-Argumentenliste*:

Teil	*Beschreibung*
BYVAL	Zeigt an, daß das Argument als Wert und nicht als nahe Referenz (voreingestellt) übergeben wird.
SEG	Übergibt das Argument als Segment- (lange) Adresse.
Argument	Eine BASIC-Variable, Datenfeld oder Konstante, die an die Prozedur übergeben wird.

CALLS hat die gleiche Wirkung wie die Benutzung von **CALL** mit **SEG** vor jedem Argument: Jedes Argument in der **CALLS**-Anweisung wird als Segmentadresse übergeben.

Hinweis Die oben beschriebene **CALL**- und **CALLS**-Syntax ruft eine BASIC-Prozedur nicht korrekt auf – nur Prozeduren in anderen Sprachen. Sehen Sie den separaten Eintrag zu **CALL (BASIC-Prozeduren)**.

Wenn die Argumentenliste einer der Anweisungen ein Datenfeldargument enthält, wird das Datenfeld mit dem Datenfeldnamen und einem Klammernpaar angegeben:

```
DIM ZahlenDatenfeld(20) AS INTEGER
.
.
.
CALL ShellSort(ZahlenDatenfeld() AS INTEGER)
```

Wenn Sie die **CALL**-Anweisung benutzen, ist das **CALL**-Schlüsselwort wahlfrei. Wenn Sie **CALL** weglassen, müssen Sie die Prozedur in einer **DECLARE**-Anweisung deklarieren. Beachten Sie bitte, daß auch die Klammern der Argumentenliste weggelassen werden, wenn Sie **CALL** weglassen. In Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie weitere Informationen, wie Prozeduren ohne das Schlüsselwort **CALL** aufgerufen werden.

Das Ergebnis des **BYVAL**-Schlüsselwortes unterscheidet sich von der Übergabe als Wert in BASIC:

```
CALL Differenz (BYVAL A, (B))
```

Für das erste Argument wird nur der *Wert* von `A` an `Differenz` übergeben. Im Gegensatz dazu wird `(B)` berechnet, ein temporärer Bereich für den Wert angelegt, und die *Adresse* des temporären Bereichs an `Differenz` übergeben. Sie können die BASIC Wert-Übergabe für ein Argument benutzen, müssen aber die Prozedur der anderen Sprache so schreiben, daß die Prozedur eine Adresse übernimmt.

Hinweis Wenn sich *Name* auf eine Assembler-Unterroutine bezieht, so muß er ein **PUBLIC**-Name (Symbol) sein.

Bitte beachten Sie, daß **PUBLIC**-Namen, die mit "$" und "_" beginnen, mit Namen, die vom BASIC-Laufzeitsystem verwendet werden, in Konflikt geraten können. Doppelt vergebene Namen führen beim Binden zu einer Linker-Fehlermeldung (`Symbol bereits definiert`).

Seien Sie vorsichtig, wenn Sie das **SEG**-Schlüsselwort zur Übergabe von Datenfeldern verwenden, da BASIC u.U. Variablen im Speicher bewegt, bevor die aufgerufene Unterroutine mit der Ausführung beginnt. Alles in einer Argumentenliste, was Speicherbewegungen verursacht, kann zu Problemen führen. Sie können sicherer Variablen mit **SEG** übergeben, wenn die Argumentenliste der **CALL**-Anweisung nur einfache Variablen, arithmetische Ausdrücke oder indizierte Datenfelder ohne Verwendung eingebauter oder benutzerdefinierter Funktionen enthält.

Eine Liste von Ereignissen, die die Bewegung von Variablen verursachen, und Informationen darüber, wann kurze oder lange Adressen für Variablen zu verwenden sind, finden Sie in Abschnitt 2.3.3, "Speicherzuweisung für Variablen".

Unterschiede zu BASICA

Von BASICA aufgerufene Assembler-Programme, die Zeichenkettenargumente haben, müssen geändert werden, weil der Zeichenkettenbeschreiber jetzt 4 Bytes lang ist. Diese vier Bytes sind das niederwertige und das höherwertige Byte der Länge, gefolgt vom niederwertigen und höherwertigen Byte der Adresse.

Um die aufzurufende Routine zu finden, benutzt die **BASICA**-Anweisung **CALLS** die Segmentadresse, die durch die zuletzt ausgeführte **DEF SEG**-Anweisung definiert wurde. Es gibt keinen Grund, **DEF SEG** mit der **CALLS**-Anweisung zu benutzen, da alle Argumente als lange (Segment-) Adressen übergeben werden.

Vergleichen Sie auch

CALL (BASIC), DECLARE (BASIC), DECLARE (Nicht-BASIC)

Beispiel

Vergleichen Sie das Beispiel für **VARPTR.**

CALL ABSOLUTE-Anweisung

Funktion

Übergibt die Steuerung an eine Prozedur in Maschinensprache.

Syntax

CALL ABSOLUTE ([*Argumentenliste,*]*Ganzzahlvariable*)

Anmerkungen

Die **CALL ABSOLUTE**-Anweisung hat folgende Argumente:

Argument	*Beschreibung*
Argumentenliste	Wahlfreie Argumente, die an eine Prozedur in Maschinensprache übergeben werden.
Ganzzahlvariable	Eine ganzzahlige Variable mit einem Wert, welcher der Offset vom Anfang des aktuellen Code-Segments, gesetzt durch **DEF SEG**, bis zur Anfangsadresse der Unterroutine im Speicher ist. Das Argument *Ganzzahlvariable* wird nicht an die Unterroutine übergeben. Ihr Programm sollte eine **DEF SEG**-Anweisung vor **CALL ABSOLUTE** ausführen, um das Code-Segment für die aufgerufene Routine zu setzen. Die Zuweisung eines nicht-ganzzahligen Wertes an *Ganzzahlvariable* führt zu unvorhersehbaren Ergebnissen.

Argumente in *Argumentenliste* werden an das Programm in Maschinensprache als Offsets (near pointers) vom aktuellen Datensegment übergeben. Obwohl Argumente als Offsets übergeben werden, wird das Programm in Maschinensprache mit einem "far call" aufgerufen.

Hinweis **CALL ABSOLUTE** ist dazu gedacht, Kompatibilität mit früheren BASIC-Versionen zu gewährleisten. Die Erweiterungen der **CALL**-Anweisung, zusammen mit der neuen **DECLARE**-Anweisung, bieten einen einfacheren Weg, die Assemblersprache zusammen mit BASIC zu benutzen. Weitere Informationen zur Programmierung mit verschiedenen Programmiersprachen finden Sie in Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC*.

Um **CALL ABSOLUTE** zu benutzen, müssen Sie QuickBASIC mit der korrekten Quick-Bibliothek starten oder Ihr Programm mit **QB.LIB** binden. Im Disketteninhaltsverzeichnis finden Sie weitere Informationen, wo sich diese Dateien befinden.

Unterschiede zu BASICA

Von BASICA aufgerufene Assembler-Programme, die Zeichenkettenargumente haben, müssen geändert werden, weil der Zeichenkettenbeschreiber jetzt 4 Bytes lang ist. Diese vier Bytes sind das niederwertige und das höherwertige Byte der Länge, gefolgt vom niederwertigen und höherwertigen Byte der Adresse.

Vergleichen Sie auch

CALL, CALLS (Nicht-BASIC)

Beispiel

Das folgende Beispiel benutzt **CALL ABSOLUTE**, um ein in Maschinensprache geschriebenes Programm, das in einem Datenfeld gespeichert ist, auszuführen.

```
'Dieses Programm gibt eine Nachricht aus, die anzeigt,
'ob ein mathematischer Koprozessor installiert ist
'oder nicht.
'Es verwendet ein in einem Datenfeld gespeichertes
'Programm in Maschinensprache, um die Informationen
'vom Betriebssystem zu erhalten.
DEFINT A-Z
DIM AsmProg(1 TO 7)
'Das Maschinensprache-Programm, gespeichert als Daten
'zum Einlesen in das Datenfeld.
DATA &H55                :'PUSH BP        Speichere
                          '               Basiszeiger.
DATA &H8B, &HEC          :'MOV  BP,SP     Hole unseren
                          '               eigenen.
DATA &HCD, &H11          :'INT  11H       Führe ROM-BIOS-
                          '               Aufruf durch.
DATA &H8B, &H5E, &H06 :'MOV  BX,6[BP] Hole Argument-
                          '               Adresse.
DATA &H89, &H07          :'MOV  [BX],AX   Speichere Liste
                          '               in Argument.
DATA &H5D                :'POP  BP        Stelle
                          '               Basiszeiger
                          '               wieder her.
```

```
DATA &HCA, &H00, &H02 :'RET  2        Hole Argument
                       '              vom Stapel und
                       '              führe langen
                       '              Rücksprung aus.
'Hole den Start-Offset des Datenfeldes.
P=VARPTR(AsmProg(1))
'Setze das Maschinensprache-Programm in das Datenfeld.
FOR I=0 TO 13
   READ J
   POKE (P+I),J
NEXT I
'Führe das Programm aus. Das Programm erwartet ein
'einzelnes Ganzzahlargument.
DEF SEG=VARSEG(AsmProg(1))  'Wechsele das Segment.
CALL ABSOLUTE (X%,VARPTR(AsmProg(1)))
DEF SEG    'Stelle das Segment wieder her.
'X% enthält jetzt das von DOS angegebene Bit-Muster
'der Ausstattungsliste.
'Maskiere alle Bits bis auf das Koprozessor-Bit (Bit2) aus.
CoProzessor=X% AND &H0002
'Gib die passende Meldung aus.
IF CoProzessor=2 THEN
   PRINT "Mathematischer Koprozessor vorhanden."
ELSE
   PRINT "Kein mathematischer Koprozessor."
END IF
END
```

CALL INT86OLD-Anweisungen

Funktion

Erlaubt Programmen, DOS-Systemaufrufe auszuführen.

Syntax

CALL INT86OLD (*Int_nr*,*In_Datenfeld*(),*Aus_Datenfeld*())

CALL INT86XOLD (*Int_nr*,*In_Datenfeld*(),*Aus_Datenfeld*())

Anmerkungen

Hinweis Die Anweisung **CALL INTERRUPT** bietet eine einfachere Möglichkeit, DOS-Systemaufrufe auszuführen. Weitere Informationen finden Sie in der Beschreibung von **CALL INTERRUPT**.

Die folgende Liste beschreibt die Argumente von **INT86OLD** und **INT86XOLD**:

Argument	***Beschreibung***
Int_nr	Der auszuführende DOS-Interrupt. Der Interrupt ist eine Ganzzahl zwischen 0 und 255. In Ihren DOS-Unterlagen finden Sie die Interrupt-Nummern.
In_Datenfeld()	Ein ganzzahliges Datenfeld, das die Registerwerte festlegt, wenn der Interrupt ausgeführt wird. **INT86OLD** verwendet ein Datenfeld mit 8 Elementen. **INT86XOLD** verwendet ein Datenfeld mit 10 Elementen. Die Tabelle weiter unten führt die Datenfeldelemente und die entsprechenden Register auf.
Aus_Datenfeld(*y*)	Enthält die Registerwerte nach einem Interrupt. Es hat die gleiche Struktur wie *In_Datenfeld*.

Wenn ein Fehler auftritt, bleiben *Int_nr* = -1 und Werte von *Aus_Datenfeld* unverändert. Fehler werden verursacht, wenn *Int_nr* nicht im Bereich 0 - 255 liegt.

Datenfeldelement	***Register***
In_Datenfeld(*x*)	**AX**
In_Datenfeld(*x+1*)	**BX**
In_Datenfeld(*x+2*)	**CX**
In_Datenfeld(*x+3*)	**DX**
In_Datenfeld(*x+4*)	**BP**
In_Datenfeld(*x+5*)	**SI**
In_Datenfeld(*x+6*)	**DI**
In_Datenfeld(*x+7*)	**FLAGS**
In_Datenfeld(*x+8*)*	**DS**
In_Datenfeld(*x+9*)*	**ES**

* Diese Datenfeldelemente werden nur bei INT86XOLD benutzt. Um die aktuellen Laufzeitwerte von DS und ES zu verwenden, weisen Sie den Datenfeldelementen 8 und 9 den Wert -1 zu.

Die **INT86OLD**- und die **INT86XOLD**-Routinen ändern alle Register außer **BP** und **DS**.

INT86OLD und **INT86XOLD** bieten Kompatibilität zu älteren Programmen, die **INT86** und **INT86X** benutzen. Wie **INT86** und **INT86X** werden **INT86OLD** und **INT86XOLD** in einer Quick-Bibliothek mit den Originaldisketten geliefert. Die Disketten enthalten auch eine Header-Datei zum Umgang mit diesen Prozeduren. Weitere Informationen finden Sie in der Liste der Disketteninhalte. In Kapitel 8, "Quick-Bibliotheken", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen zu Quick-Bibliotheken.

Beachten Sie, daß **INT86OLD** und **INT86XOLD** die Verwendung von **VARPTR** nicht erfordern. Die Registerwerte werden in den Datenfeldern, beginnend mit dem ersten Datenfeldelement, gespeichert.

Beispiel

Das folgende Beispiel verwendet **INT86OLD**, um eine Datei zu öffnen und Text in die Datei zu schreiben:

```
' Header-Datei für INT86OLD, etc. aufnehmen
' $INCLUDE:'QB.BI'
DIM INDAT%(7), AUSDAT%(7)  'Definiere Ein- und
                           'Ausgabedatenfelder für
                           'INT86
' Definiere Register-Datenfeldindizes, um das Programm
' verständlicher zu machen
CONST AX=0, BX=1, CX=2, DX=3, BP=4, SI=5, DI=6, FL=7
INDAT%(AX) = &H3C00       'DOS-Funktion zum Erstellen
                          'einer Datei
INDAT%(CX) = 0            'DOS-Attribut für angelegte
                          'Datei
INDAT%(DX) = SADD("FOO.TXT"+CHR$(0))
                          'Zeiger auf Zeichenkette des
                          'Dateinamens mit Null-Byte als
                          'Ende
CALL INT86OLD(&H21,INDAT%(),AUSDAT%())
                          'Führe die Erstellung durch
INDAT%(BX) = AUSDAT%(AX)  'Bewege die erstellte
                          'Dateibehandlung zum
                          'Schreiben
INDAT%(AX) = &H4000 'DOS-Funktion zum Schreiben in
                    'Datei
TEXT$ = "Hallo Seattle"+CHR$(13)+CHR$(10)
          'Definiere Text zum Schreiben in die Datei
```

```
INDAT%(CX) = LEN(TEXT$)   'Ermittle Länge der
                          'Zeichenkette TEXT
INDAT%(DX) = SADD(TEXT$)  'Ermittle Adresse der
                          'Zeichenkette TEXT
CALL INT86OLD(&H21,INDAT%(),AUSDAT%())
                      ' Schreiben ausführen
INDAT%(AX) = &H3E00   'DOS-Funktion zum Schließen einer
                      'Datei
CALL INT86OLD(&H21,INDAT%(),AUSDAT%())
                          ' Schließen ausführen
```

CALL INTERRUPT-Anweisungen

Funktion

Erlaubt BASIC-Programmen, DOS-Systemaufrufe auszuführen.

Syntax

CALL INTERRUPT (*Interruptnum, Inreg, Ausreg*)

CALL INTERRUPTX (*Interruptnum, Inreg, Ausreg*)

Anmerkungen

Die folgende Liste beschreibt die Argumente für die **CALL INTERRUPT**- und **CALL INTERRUPTX**-Anweisungen:

Argument	*Beschreibung*
Interruptnum	Die DOS-Interrupt-Nummer. Die Interrupt-Nummer ist eine Ganzzahl zwischen 0 und 255. In Ihren DOS-Unterlagen finden Sie weitere Informationen über Interrupts.
Inreg	Die Variable *Inreg* enthält die zur Ausführung des Interrupts verwendeten Registerwerte. Sie wird als Typ `RegTyp` deklariert. Unten finden Sie eine Beschreibung des benutzerdefinierten Typs `RegTyp`.
Ausreg	Die Variable *Ausreg* enthält die Werte der Register nach der Ausführung des Interrupts. Sie wird als Typ `RegTyp` deklariert. Unten finden Sie eine Beschreibung des benutzerdefinierten Typs `RegTyp`.

Die Anweisungen **CALL INTERRUPT** und **CALL INTERRUPTX** ersetzen die Routinen **INT86** und **INT86X**, die in früheren BASIC-Versionen verwendet wurden. Sie bieten eine komfortablere Möglichkeit, DOS-Interrupts und DOS-Dienste zu nutzen.

Die Registerwerte vor und nach dem Interrupt werden in Variablen übergeben, die als Typ `RegTyp` deklariert werden. Die folgende Anweisung definiert den benutzerdefinierten Typ `RegTyp`:

```
TYPE RegTyp
   AX AS INTEGER
   BX AS INTEGER
   CX AS INTEGER
   DX AS INTEGER
   BP AS INTEGER
   SI AS INTEGER
   DI AS INTEGER
   FLAGS AS INTEGER
   DS AS INTEGER
   ES AS INTEGER
END TYPE
```

Jedes Element dieses Typs korrespondiert mit einem CPU-Register.

INTERRUPTX benutzt die Werte in den **DS**- und **ES**-Registern. Um die aktuellen Werte dieser Register zu benutzen, setzen Sie die Verbundelemente auf -1.

CALL INTERRUPT und **CALL INTERRUPTX** werden in einer Quick-Bibliothek auf Ihren Disketten geliefert. Auf den Disketten befindet sich ebenfalls eine Header-Datei, die die notwendigen Deklarationen zur Verwendung dieser Prozeduren enthält. Sehen Sie im Disketteninhaltsverzeichnis für weitere spezifische Informationen. In Kapitel 8, "Quick-Bibliotheken", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen zu Quick-Bibliotheken.

Beispiel

Das folgende Programm benutzt **CALL INTERRUPT**, um die Attributenliste einer Datei so zu verändern, daß die Datei bei der Verwendung des DOS-Befehls **DIR** nicht erscheint:

```
DECLARE SUB TestFehler (AXReg%, flags%)
' $INCLUDE: 'QB.BI'
DEFINT A-Z
DIM EinRegs AS RegTyp, AusRegs AS RegTyp
' Hole Dateinamen und auszuführende Aktion.
CLS
PRINT "Programm Verstecke Datei": PRINT
```

```
INPUT "Geben Sie den Dateinamen ein: ", DateiName$
DO
   INPUT "Verstecken oder Hervorholen (V or H): ", Aktion$
   Aktion$ = UCASE$(Aktion$)
LOOP WHILE Aktion$ <> "V" AND Aktion$ <> "H"
' Hänge für die DOS-Funktion an das Ende der
' Zeichenkette ein Null-Byte an.
DateiName$ = DateiName$ + CHR$(0)
' Hole das aktuelle Dateiattribut.
' Aktuelles Attribut kommt in AusRegs.AX zurück.
EinRegs.ax = &H4300
EinRegs.dx = SADD(DateiName$)
CALL INTERRUPT(&H21, EinRegs, AusRegs)
CALL TestFehler(AusRegs.ax, AusRegs.flags)
' Wandele das Attributbit Versteckt in den alten
' Attributwert um.
IF Aktion$ = "H" THEN
   EinRegs.cx = AusRegs.cx AND &HFD
ELSE
   EinRegs.cx = AusRegs.cx OR &H2
END IF
' Setze AX, um die DOS-Funktion Verändere Attribut
' anzuzeigen.
EinRegs.ax = &H4301
CALL INTERRUPT(&H21, EinRegs, AusRegs)
CALL TestFehler(AusRegs.ax, AusRegs.flags)
END
' Wenn Carry-Flag gesetzt ist , schreibe
' Fehlermeldung und beende das Programm.
SUB TestFehler (AXReg, flags) STATIC
   IF (&H1 AND flags) <> 0 THEN
      ' Hole die Fehlernummer aus AX.
      SELECT CASE AXReg AND &HF
         CASE 2
            PRINT "Datei nicht gefunden."
         CASE 3
            PRINT "Pfad nicht gefunden."
         CASE 5
            PRINT "Zugriff verweigert."
         CASE ELSE
            PRINT "Unbekannter Fehler."
      END SELECT
      END
   END IF
END SUB
```

CDBL-Funktion

Funktion

Wandelt einen numerischen Ausdruck in eine Zahl doppelter Genauigkeit um.

Syntax

CDBL(*Numerischer Ausdruck*)

Anmerkungen

Numerischer Ausdruck kann jeder beliebige numerische Ausdruck sein.

Diese Funktion bewirkt dasselbe wie die Zuweisung des numerischen Ausdrucks an eine Variable doppelter Genauigkeit.

Bitte beachten Sie, daß die Ergebnisse der **CDBL**-Funktion nicht genauer sind als die des ursprünglichen Ausdrucks. Die zusätzlichen Nachkommastellen sind nicht bedeutend, solange der Ausdruck nicht mit doppelter Genauigkeit berechnet wird.

Beispiel

Das folgende Beispiel zeigt, wie die Genauigkeit des numerischen Ausdrucks die Ergebnisse der **CDBL**-Funktion beeinflussen:

```
X = 7/9
X# = 7/9
PRINT X
'Sowohl X# als auch CDBL(X) werden nur mit einer
'Genauigkeit von 7 Stellen angegeben, weil 7/9 mit
'einfacher Genauigkeit berechnet wird.
PRINT X#
PRINT CDBL(X)
'Genau bis 15 Dezimalstellen.
PRINT 7#/9#
```

Ausgabe

```
.7777778
.7777777910232544
.7777777910232544
.7777777777777778
```

CHAIN-Anweisung

Funktion

Übergibt die Steuerung aus dem aktuellen Programm an ein anderes Programm.

Syntax

CHAIN *Dateiangabe*

Anmerkungen

Das Argument *Dateiangabe* ist ein Zeichenkettenausdruck zur Identifikation des Programms, an das die Steuerung übergeben wird. Die *Dateiangabe* kann eine Pfadbeschreibung enthalten. Programme, die innerhalb der QuickBASIC-Umgebung laufen, nehmen – falls keine Erweiterung angegeben wurde – die Erweiterung **.BAS** an und können nicht zu .EXE-Dateien verkettet werden. Programme, die außerhalb der Umgebung kompiliert und ausgeführt wurden, nehmen die Erweiterung .EXE an und können nicht mit Quelldateien verkettet werden.

Um Variablen zwischen Programmen zu übergeben, können Sie die **COMMON**-Anweisung verwenden, um einen unbenannten **COMMON**-Block zu vereinbaren. Vergleichen Sie auch die Beschreibung der **COMMON**-Anweisung.

Wenn Sie ein Programm außerhalb der BASIC-Umgebung kompilieren (d.h. nicht im Speicher), beachten Sie, daß die Bibliothek **BCOM40.LIB** die Verwendung der **COMMON**-Anweisung nicht unterstützt. Um **COMMON** mit verketteten Programmen außerhalb der Umgebung zu benutzen, verwenden Sie die voreingestellte **BRUN40.EXE**, indem Sie das Programm unter Benutzung der Option "EXE erfordert BRUN40.EXE" in dem Dialogfeld **EXE-Datei Erstellen** kompilieren; oder verwenden Sie die voreingestellte **BRUN40.LIB**, indem Sie *ohne* die **/o** Option kompilieren.

Hinweis Wenn Programme **BRUN40.LIB** benutzen, bleiben die Dateien während der Verkettung offen, es sei denn, sie werden mit einer **CLOSE**-Anweisung explizit geschlossen.

Unterschiede zu BASICA

BASICA setzt die Erweiterung **.BAS** voraus. QuickBASIC setzt eine der beiden Erweiterungen – **.BAS** oder **.EXE** – voraus, abhängig davon, ob das Programm innerhalb der Umgebung läuft oder ob es außerhalb der Umgebung kompiliert und ausgeführt wird. Wenn Sie keine Dateierweiterung angeben, arbeitet **CHAIN** in QuickBASIC und BASICA gleich.

BASIC unterstützt nicht die bei BASICA verfügbaren Optionen **ALL, MERGE, DELETE** oder die Spezifikation einer Zeilennummer.

Ohne die Option Zeilennummer startet die Ausführung am Beginn des verketteten Programmes. Deshalb kann ein verkkettetes Programm, das zu einem unachtsam geschriebenen verkettenden Programm zurückkehrt, eine Endlos-Schleife bewirken.

Vergleichen Sie auch

CALL (BASIC), COMMON

Beispiel

Siehe auch das Beispiel für die **COMMON**-Anweisung.

CHDIR-Anweisung

Funktion

Ändert das aktuelle Standardverzeichnis für das angegebene Laufwerk.

Syntax

CHDIR *Verzeichnisangabe*

Anmerkungen

Verzeichnisangabe ist ein Zeichenkettenausdruck, der den Namen des Verzeichnisses angibt, das das aktuelle Verzeichnis werden soll. Die *Verzeichnisangabe* muß weniger als 64 Zeichen lang sein und hat folgende Syntax:

[*Laufwerk:*][\] *Verzeichnis* [\ *Verzeichnis*]...

Das Argument *Laufwerk:* ist eine wahlfreie Laufwerksangabe. Wenn Sie *Laufwerk:* nicht angeben, ändert **CHDIR** das voreingestellte Verzeichnis im aktuellen Laufwerk.

CHDIR unterscheidet sich in zwei Punkten von der DOS-Anweisung **CHDIR:**

1. Die BASIC-Anweisung kann nicht auf **CD** gekürzt werden.
2. Es gibt keine Form der **CHDIR**-Anweisung, die das aktuelle Verzeichnis angibt.

Hinweis **CHDIR** wechselt das voreingestellte Verzeichnis, nicht das voreingestellte Laufwerk. Zum Beispiel: Wenn das voreingestellte Laufwerk C ist, ändert die folgende **CHDIR**-Anweisung das voreingestellte Verzeichnis auf Laufwerk D, das voreingestellte Laufwerk ist jedoch immer noch C:

```
CHDIR "D:TMP"
```

Vergleichen Sie auch

MKDIR, RMDIR

Beispiele

```
'Macht \INLAND\VERKAUF zum aktuellen Verzeichnis auf
'dem Standard-Laufwerk.
CHDIR "\INLAND\VERKAUF"
'Ändert das aktuelle Verzeichnis auf Laufwerk B in
'BENUTZER; das Standard-Laufwerk wird jedoch nicht
'in B geändert.
CHDIR "B:BENUTZER"
```

CHR$-Funktion

Funktion

Gibt eine aus einem Zeichen bestehende Zeichenkette zurück, die dem im Argument angegebenen ASCII-Code entspricht.

Syntax

CHR$ (*Code*)

Anmerkungen

**CHR$** wird in der Regel benutzt, um ein Sonderzeichen an den Bildschirm oder den Drucker zu senden. Beispielsweise können Sie einen Formularvorschub (CHR$(12)) ausgeben, um den Bildschirm zu löschen und den Cursor zur Grundstellung zurückzubringen.

CHR$ kann außerdem verwendet werden, um ein doppeltes Anführungszeichen (") in eine Zeichenkette einzufügen:

```
Mldg$ = CHR$(34)+ "eingeschlossene Zeichenkette" + CHR$(34)
```

Diese Zeile fügt ein doppeltes Anführungszeichen an den Beginn und an das Ende einer Zeichenkette.

Vergleichen Sie auch

ASC; Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes"

Beispiel

Das folgende Beispiel benutzt **CHR$**, um Grafikzeichen des erweiterten Zeichensatzes darzustellen.

```
DEFINT A-Z
'Zeige zwei doppelt umrandete Rechtecke.
CALL DRecht (5,22,18,40)
CALL DRecht (1,4,4,50)
END
'Unterroutine, um Rechtecke auf dem Bildschirm zu
'zeigen.
'
'Parameter:
'OZeile%,OSpalte% : Zeile und Spalte der oberen linken
'Ecke.
'USpalte%,UZeile% : Zeile und Spalte der unteren
'rechten Ecke.
'Verwende Konstanten für die Grafikzeichen des
'erweiterten Zeichensatzes.
CONST OLINKS=201, ORECHTS=187, VERTIKAL=186
CONST HORIZONTAL=205, ULINKS=200, URECHTS=188
SUB DRecht (OZeile%,OSpalte%,UZeile%,USpalte%) STATIC
   'Zeichne den obersten Teil des Rechtecks,
   'beginnend mit der oberen linken Ecke.
   LOCATE OZeile%,OSpalte% : PRINT CHR$(OLINKS);
   LOCATE ,OSpalte%+1
   PRINT STRING$(USpalte%-OSpalte%,CHR$(HORIZONTAL));
   LOCATE ,USpalte% : PRINT CHR$(ORECHTS);
```

```
'Zeichne die Seiten des Rechtecks.
FOR I=OZeile%+1 TO UZeile%-1
   LOCATE I,OSpalte% : PRINT CHR$(VERTIKAL);
   LOCATE  ,USpalte% : PRINT CHR$(VERTIKAL);
NEXT I
'Zeichne den untersten Teil des Rechtecks.
LOCATE UZeile%, OSpalte% : PRINT CHR$(ULINKS);
LOCATE ,OSpalte%+1
PRINT STRING$(USpalte%-OSpalte%,CHR$(HORIZONTAL));
LOCATE ,USpalte% : PRINT CHR$(URECHTS);
END SUB
```

CINT-Funktion

Funktion

Wandelt einen numerischen Ausdruck in eine ganze Zahl um, indem die Stellen hinter dem Komma gerundet werden.

Syntax

CINT (*Numerischer Ausdruck*)

Anmerkungen

Falls der *Numerischer Ausdruck* nicht im Bereich von -32.768 bis 32.767 liegt, produziert die Funktion eine `Überlauf` Laufzeit-Fehlermeldung.

CINT unterscheidet sich von den Funktionen **FIX** und **INT**, die die Nachkommastellen nicht runden, sondern abschneiden. Im Beispiel für die **INT**-Funktion finden Sie die Unterschiede zwischen diesen Funktionen erläutert.

Vergleichen Sie auch

CDBL, CSNG, FIX, INT

Beispiel

Das folgende Beispiel konvertiert einen Winkel in Bogenmaß in einen Winkel in Grad und Minuten um:

```
'Setze Konstanten zur Konvertierung von Bogenmaß in
'Grad.
CONST PI=3.141593, BOGINGR=180./PI
INPUT "Winkel im Bogenmaß = ",Winkel 'Lies den Winkel
                                      'im Bogenmaß.
Winkel = Winkel * BOGINGR 'Wandle Bogenmaß in Grad um.
Min = Winkel - INT(Winkel)'Berechne Stellen hinter
                          'dem Komma.
'Wandle Bruchteil in Wert zwischen 0 und 60 um.
Min = CINT(Min * 60)
Winkel = INT(Winkel)         'Hole ganzzahligen Teil.
IF Min = 60 THEN             '60 Minuten = 1 Grad.
   Winkel = Winkel + 1
   Min = 0
END IF
PRINT "Winkel gleich" Winkel "Grad" Min "Minuten"
```

Ausgabe

```
Winkel im Bogenmaß = 1.5708
Winkel gleich 90 Grad 0 Minuten
```

CIRCLE-Anweisung

Funktion

Zeichnet eine Ellipse oder einen Kreis mit dem angegebenen Mittelpunkt und Radius.

Syntax

CIRCLE [**STEP**] (*x,y*) , *Radius* [,[*Farbe*] [,[*Anfang*][,[*Ende*][,*Aspekt*]]]]

Anmerkungen

Die folgende Liste beschreibt die Teile der **CIRCLE**-Anweisung:

Teil	*Beschreibung*
STEP	Die Option **STEP** gibt an, daß *x* und *y* Offsets relativ zur aktuellen Grafik-Cursorposition sind.
x,y	Die *x*- und y-Koordinaten für den Mittelpunkt des Kreises oder der Ellipse.
Radius	Der Radius des Kreises oder der Ellipse im aktuellen Koordinatensystem.
Farbe	Das Attribut der gewünschten Farbe. Für weitere Informationen siehe hierzu die Beschreibung der **COLOR**- und **SCREEN**-Anweisungen. Die Standardfarbe ist die Textfarbe.
Anfang, Ende	Der *Anfangs*- und *End*winkel im Bogenmaß für den zu zeichnenden Kreisbogen. Die Argumente *Anfang* und *Ende* werden benutzt, um Teilkreise oder Ellipsen zu malen. Diese Argumente können einen Wert zwischen -2π rad (Bogenmaß) und 2π rad haben, mit π = 3,141593. Der Standardwert für *Anfang* ist 0 rad. Der Standardwert für *Ende* ist 2π rad. Wenn der *Anfangs*- oder *End*winkel negativ ist, zeichnet **CIRCLE** einen Radius zu diesem Punkt auf dem Kreisbogen und betrachtet den Winkel als positiv. Der Anfangswinkel kann kleiner als der Endwinkel sein. Wenn Sie *Ende*, aber nicht *Anfang*, angeben, wird der Bogen von 2π bis *Ende* gezeichnet. Geben Sie dagegen *Anfang*, nicht aber *Ende* an, so zeichnet diese Anweisung den ganzen Kreisbogen.
Aspekt	Der Aspekt (Seitenverhältnis) oder das Verhältnis des y- zum x-Radius. Der Standardwert für *Aspekt* ist der zum Zeichnen eines Kreises im gegebenen Bildschirmmodus erforderliche Wert. Dieser Wert wird wie folgt berechnet: **4 * (*y-Bildpunkte* / *x-Bildpunkte*) / 3** wobei *x-Bildpunkte* x *y-Bildpunkte* die Bildschirmauflösung ist. Zum Beispiel ist in **SCREEN 1**, mit einer Auflösung von 320 x 200, die Vorgabe für *Aspekt* 4* (200/320)/3 oder 5/6. Ist der Aspekt kleiner als eins, so ist *Radius* der *x*–Radius. Ist er größer als eins, so ist *Radius* gleich dem y-Radius.

Hinweis Um einen Radius zum Winkel 0 zu zeichnen (eine horizontale Linie nach rechts), geben Sie den Winkel bitte nicht als -0 an; benutzen Sie stattdessen einen sehr kleinen Wert ungleich Null:

```
Zeichnet eine Viertelecke einer Kreisgrafik:
SCREEN 2
CIRCLE (200,100),60,,-.0001,-1.57
```

Sie können ein Argument in der Mitte der Anweisung weglassen, müssen jedoch die Kommata setzen. In der folgenden Anweisung wurde das Argument *Farbe* ausgelassen:

```
CIRCLE STEP (150,200), 94,,0.0,6.28
```

Wenn Sie das letzte Argument auslassen, brauchen Sie das entsprechende Komma nicht zu setzen.

Der letzte Punkt, auf den sich **CIRCLE** nach dem Zeichnen einer Ellipse oder eines Kreises bezieht, ist deren bzw. dessen Mittelpunkt.

Sie können Koordinaten, die außerhalb des Bildschirms oder des Darstellungsfeldes liegen, benutzen.

Sie können Koordinaten als Absolutwerte darstellen oder die Option **STEP** verwenden, um die Position des Mittelpunktes in Relation zum vorhergehenden Bezugspunkt zu zeigen. Ist beispielsweise der vorhergehende Bezugspunkt (10,10), so bewirkt die nachstehende Anweisung, daß ein Kreis mit dem Radius 75 gezeichnet wird, dessen Mittelpunkt 10 von der aktuellen *x*-Koordinate und 5 von der aktuellen *y*-Koordinate versetzt ist; d. h. der Kreis hat dann in den aktuellen Koordinaten den Mittelpunkt (20,15):

```
CIRCLE STEP (10,5), 75
```

Beispiel

Das folgende Programm zeichnet zunächst einen Kreis mit fehlendem oberen linken Viertel. Danach werden relative Koordinaten benutzt, um einen zweiten Kreis in dem fehlenden Kreisviertel zu positionieren. Zum Schluß wird ein anderer Aspekt benutzt, um eine kleine Ellipse innerhalb des kleinen Kreises zu zeichnen.

```
CONST PI=3.141593
SCREEN 2
'Zeichne einen Kreis mit einem fehlenden linken oberen
'Viertel.
'Verwende negative Zahlen, sodaß die Radien gezeichnet
'werden.
CIRCLE (320,100), 200,, -PI, -PI/2
```

```
'Benutze relative Koordinaten, um einen Kreis im
'fehlenden Viertel zu zeichnen.
CIRCLE STEP (-100,-42),100
'Zeichne eine kleine Ellipse innerhalb des
'Kreises.
CIRCLE STEP (0,0), 100,,,, 5/25
'Zeige das Bild, bis eine Taste betätigt wird.
LOCATE 25,1 : PRINT "Beenden mit jeder Taste.";
DO
LOOP WHILE INKEY$=""
```

CLEAR-Anweisung

Funktion

Reinitialisiert alle Programmvariablen, schließt Dateien und setzt die Stapelgröße (Stack).

Syntax

CLEAR [, , *Stapel*]

Anmerkungen

Die Anweisung **CLEAR** führt folgende Aktionen aus:

- Sie schließt alle Dateien und gibt die Dateipuffer frei.
- Sie löscht alle **COMMON**-Variablen.
- Sie setzt die numerischen Variablen und Datenfelder auf Null.
- Sie setzt alle Zeichenkettenvariablen auf Null.
- Sie reinitialisiert den Stapel und verändert wahlweise seine Größe.

Der Parameter *Stapel* reserviert Stapelplatz für Ihr Programm. QuickBASIC addiert zu der von ihm erforderlichen Größe des Stapelbereichs die Anzahl von in *Stapel* spezifizierten Bytes und setzt die Stapelgröße entsprechend diesem Ergebnis.

Hinweis Um die Kompatibilität von QuickBASIC zu BASICA zu gewährleisten, sind die zwei Kommata vor *Stapel* erforderlich. BASICA hat ein zusätzliches Argument, um die Größe des Datensegmentes zu setzen. Da QuickBASIC das Datensegment automatisch verwaltet, ist der erste Parameter nicht länger erforderlich.

Wenn Ihr Programm tief verschachtelte Unterroutinen oder Prozeduren enthält, oder wenn Sie rekursive Prozeduren benutzen, können Sie die **CLEAR**-Anweisung benutzen, um die Stapelgröße zu erhöhen. Sie müssen vielleicht auch die Stapelgröße erhöhen, wenn Ihre Prozeduren eine große Anzahl von Argumenten haben.

Das Löschen des Stapels zerstört die Rücksprungadressen, die während Ausführungen von **GOSUB** auf dem Stapel abgelegt wurden. Dies macht es unmöglich, die **RETURN**-Anweisung korrekt auszuführen und erzeugt einen `RETURN ohne GOSUB` Laufzeitfehler. Die Verwendung einer **CLEAR**-Anweisung in einer **SUB** oder einer **FUNCTION** erzeugt einen `Unzulässiger Funktionsaufruf` Laufzeitfehler.

Unterschiede zu BASICA

BASICA-Programme, die **CLEAR** benutzen, können Änderungen erfordern. In BASICA-Programmen gehen sämtliche **DEF FN**-Funktionen oder Datentypen, die mit **DEF***Typ*-Anweisungen deklariert wurden, nach einer **CLEAR**-Anweisung verloren. In kompilierten Programmen sind diese Informationen nicht verloren, da die Deklarationen während des Kompilierens festgelegt werden.

Vergleichen Sie auch

FRE, SETMEM

Beispiel

Die folgende Anweisung löscht alle Programmvariablen und setzt den Stapel auf 2000 Bytes:

```
CLEAR ,,2000
```

CLNG-Funktion

Funktion

Wandelt einen numerischen Ausdruck in eine lange (4-Byte-) Ganzzahl durch Rundung der Nachkommastellen des Ausdrucks um.

Syntax

CLNG(*Numerischer Ausdruck*)

Anmerkungen

Wenn *Numerischer Ausdruck* nicht im Bereich von -2.147.483.648 bis 2.147.483.647 liegt, gibt die Funktion die Fehlermeldung `Überlauf` aus.

Beispiel

Das folgende Beispiel zeigt, wie **CLNG** eine Zahl vor der Konvertierung rundet:

```
A=32767.45
B=32767.55
PRINT CLNG(A); CLNG(B)
```

Ausgabe

```
32767     32768
```

CLOSE-Anweisung

Funktion

Beendet Ein-/Ausgabe in/aus eine(r) Datei oder ein(em) Gerät.

Syntax

CLOSE [[#]*Dateinummer* [,[#]*Dateinummer*]...]

Anmerkungen

Die Anweisung **CLOSE** ergänzt die Anweisung **OPEN**.

Dateinummer ist die Nummer, unter der die Datei geöffnet wurde. Ein **CLOSE** ohne Argumente schließt alle offenen Dateien und deaktiviert alle aktivierten Geräte.

Die Verbindung einer Datei mit einer Dateinummer endet nach Ausführung der Anweisung **CLOSE**. Danach können Sie die Datei mit derselben oder einer anderen Dateinummer erneut öffnen. Sobald Sie eine Datei schließen, können Sie deren Nummer für jede beliebige nicht eröffnete Datei nutzen.

Eine **CLOSE**-Anweisung für eine Datei oder ein Gerät, die/das für sequentielle Ausgabe eröffnet/aktiviert wurde, schreibt den Endausgabepuffer in diese Datei oder dieses Gerät.

CLOSE gibt alle mit der/den geschlossenen Datei/en verknüpften Pufferbereiche frei.

Die Anweisungen **CLEAR, END, RESET, RUN** und **SYSTEM** schließen immer alle Dateien automatisch.

Beispiel

Siehe Beispiel für **CALL-Anweisung (BASIC-Prozeduren)**.

CLS-Anweisung

Funktion

Löscht den Bildschirm.

Syntax

CLS [{0 | 1 | 2}]

Anmerkungen

CLS kann auf verschiedene Arten verwendet werden, wie die folgende Liste beschreibt:

Anweisung	*Beschreibung*
CLS 0	Löscht den gesamten Text und alle Grafiken vom Bildschirm.
CLS 1	Löscht nur das Grafik-Darstellungsfeld, wenn die **VIEW**-Anweisung ausgeführt wurde. Anderenfalls löscht **CLS 1** den gesamten Bildschirm.
CLS 2	Löscht nur das Text-Darstellungsfeld und läßt die unterste Bildschirmzeile (abhängig vom Bildschirmmodus Zeile 25, 30, 43 oder 60) ungeändert.
CLS	Löscht entweder das Grafik-Darstellungsfeld oder das Text-Darstellungsfeld. Wenn das Grafik-Darstellungsfeld aktiviert ist, löscht **CLS** ohne ein Argument nur dieses Feld. Ist das Grafik-Darstellungsfeld nicht aktiviert, so löscht **CLS** das Textfeld und stellt die Anzeigezeile der Funktionstasten (die unterste Bildschirmzeile) wieder her.

Außerdem bringt **CLS** den Cursor in die Grundstellung in der oberen linken Bildschirmecke zurück.

Vergleichen Sie auch

VIEW, VIEW PRINT, WINDOW

Beispiel

Das folgende Programm zeichnet Zufallskreise in ein Grafikfenster und schreibt Text in ein Textfenster. Das Grafikfenster wird gelöscht, nachdem 30 Kreise gezeichnet sind. Nachdem 45mal geschrieben wurde, löscht das Programm das Textfenster.

```
RANDOMIZE TIMER
SCREEN 1
'Richte ein Grafik-Darstellungsfeld mit Rand ein.
VIEW (5,5)-(100,80),3,1
'Richte ein Text-Darstellungsfeld ein.
VIEW PRINT 12 TO 24
'Gib eine Nachricht auf dem Bildschirm außerhalb des
'Text-Darstellungsfeldes aus.
LOCATE 25,1 : PRINT "Beenden mit jeder Taste."
Zaehl=0
DO
   'Zeichne einen Kreis mit einem zufälligen Radius.
   CIRCLE (50,40),INT((35-4)*RND+5),(Zaehl MOD 4)
   'Lösche grafisches Darstellungsfeld alle 30 mal.
   IF (Zaehl MOD 30)=0 THEN CLS 1
   PRINT "Hallo Titus. ";
   'Lösche das Text-Darstellungsfeld alle 45 mal.
   IF (Zaehl MOD 45)=0 THEN CLS 2
   Zaehl=Zaehl+1
LOOP UNTIL INKEY$<>""
```

COLOR-Anweisung

Funktion

Wählt die Anzeigefarben aus.

Syntax

COLOR [*Text*] [,[*Hintergrund*][, *Rahmen*]]	Bildschirmmodus 0
COLOR [*Hintergrund*][, *Palette*]	Bildschirmmodus 1
COLOR [*Text*][, *Hintergrund*]	Bildschirmmodi 7-10
COLOR [*Text*]	Bildschirmmodi 11-13

Anmerkungen

Mit der **COLOR**-Anweisung können Sie Text- und Hintergrundfarben für die Anzeige setzen. In **SCREEN 0** kann außerdem eine Farbe für den Rahmen gewählt werden. In **SCREEN 1** kann keine Textfarbe gewählt werden, dafür kann aber eine der zwei Vierfarbpaletten für Grafikanweisungen benutzt werden. In den Bildschirmmodi 11-13 kann nur die Textfarbe gewählt werden.

Die Werte für *Text* in den Bildschirmmodi 7-10, 12 und 13 sind Attributnummern (nicht für Farbnummern) und zeigen die dem Attribut zugeordnete Farbe an. Siehe die **PALETTE**-Anweisung für eine Beschreibung dieser Attribute.

Die Anweisung **COLOR** bestimmt nicht den Bereich möglicher Farben. Die Kombination von Adapter, Bildschirm und Modi, die mit der **SCREEN**-Anweisung gesetzt sind, bestimmen den Farbbereich. Weitere Informationen finden Sie bei der Beschreibung der **SCREEN**-Anweisung.

Nachfolgend werden die unterschiedlichen Syntax-Regeln und ihre Wirkungen in unterschiedlichen Bildschirmmodi beschrieben:

Modus	***Beschreibung***
SCREEN 0	Ändert die aktuelle Standard-Text- und Hintergrundfarbe sowie den Rahmen. Die *Text*farbe muß ein ganzzahliger Ausdruck im Bereich 0 bis 31 sein. Er dient zur Festlegung der "Vordergrund"-Farbe im Textmodus, welche die Standard-Textfarbe ist. Mit den ganzen Zahlen 0 bis 15 können 16 Farben gewählt werden. Durch Addieren von 16 zu der Farbnummer kann eine blinkende Version jeder Farbe gewählt werden; beispielsweise ist eine blinkende Farbe 7 gleich 7 + 16 oder 23. Die *Hintergrund*-Farbe muß ein ganzzahliger Ausdruck im Bereich von 0 bis 7 sein und ist die Farbe des Hintergrundes für jedes Textzeichen. Blinkende Farben werden nicht unterstützt. Die *Rahmen*-Farbe ist ein ganzzahliger Ausdruck im Bereich 0 bis 15 und ist die zum Zeichnen des Bildschirmrahmens verwendete Farbe. Blinkende Farben sind nicht zulässig. Das *Rahmen*-Argument wird vom IBM Erweiterter Grafikadapter (Enhanced Graphics Adapter, EGA), IBM Video-Grafikkarte (Video Graphics Array, VGA) und IBM Mehrfarben-Grafikkarte (Multicolor Graphics Array, MCGA) nicht unterstützt.

Modus	*Beschreibung*
SCREEN 1	Im Modus 1 hat die Anweisung **COLOR** eine eindeutige Syntax, die ein *Palette*-Argument enthält, d. h. einen ungeraden oder geraden ganzzahligen Ausdruck im Bereich von 0 bis 255. Dieses Argument bestimmt die Gruppe der Farben, die bei der Anzeige von bestimmten Farbnummern zu verwenden ist. Die Standard-Farbeinstellungen für den Parameter *Palette* sind identisch zu den folgenden **PALETTE**-Anweisungen auf einem mit EGA ausgerüsteten System:

```
COLOR ,0            'Ist gleich den nächsten drei
                    'PALETTE-Anweisungen
PALETTE  1, 2       'Attribut 1 = Farbe 2 (grün)
PALETTE  2, 4       'Attribut 2 = Farbe 4 (rot)
PALETTE  3, 6       'Attribut 3 = Farbe 6 (gelb)
COLOR ,1            'Ist gleich den nächsten drei
                    'PALETTE-Anweisungen
PALETTE  1, 3       'Attribut 1 = Farbe 3 (kobaltblau)
PALETTE  2, 5       'Attribut 2 = Farbe 5 (violett)
PALETTE  3, 7       'Attribut 3 = Farbe 7 (weiß)
```

	Bitte beachten Sie, daß im Bildschirmmodus 1 eine **COLOR**-Anweisung vorherige **PALETTE**-Anweisungen aufhebt.
SCREEN 2, SCREEN 11	Falls **COLOR** in diesem Modus benutzt wird, erfolgt die Fehlermeldung `Unzulässiger Funktionsaufruf.`
SCREEN 7- SCREEN 10	In diesen Modi kann keine Rahmen-Farbe angegeben werden. Der Grafikhintergrund wird durch die *Hintergrund*-Farbnummer festgelegt, die in dem für den Bildschirmmodus passenden gültigen Bereich der Farbnummern liegen muß. Das Argument *Text* ist die Standardfarbe zum Zeichnen von Strichen. In den Bildschirmmodi 7 bis 10 ist *Text* eine Attributnummer, während *Hintergrund* eine Farbnummer ist. Weitere Einzelheiten finden Sie bei der **SCREEN**-Anweisung.
SCREEN 12, SCREEN 13	In diesen Modi kann keine Hintergrundfarbe angegeben werden. Das *Text*-Argument bestimmt die Vordergrund-Grafikfarbe. Die Farbe muß im gültigen Bereich der Farbnummern für den Bildschirmmodus liegen. Weitere Informationen finden Sie in der Beschreibung der **SCREEN**-Anweisung.

Argumente außerhalb der gültigen numerischen Bereiche führen zur Fehlermeldung `Unzulässiger Funktionsaufruf.`

Die Textfarbe kann dieselbe Farbe wie die Hintergrundfarbe sein; hierdurch werden Zeichen unsichtbar. Die Standard-Hintergrundfarbe ist für alle Hardware-Konfigurationen und alle Bildschirmmodi schwarz (Farbnummer 0).

In den Bildschirmmodi 11 - 13 können Sie die Hintergrundfarbe setzen, indem Sie mit der Anweisung **PALETTE** eine Farbe Attribut 0 zuweisen. Um beispielsweise die Farbe 8224 (leicht violett) als Hintergrundfarbe zu wählen, können Sie die folgende **PALETTE**-Anweisung benutzen:

```
PALETTE 0, 8224
```

Wenn eine EGA, VGA oder MCGA-Karte installiert ist, bietet Ihnen die Anweisung **PALETTE** die Möglichkeit, bei der Zuweisung von verschiedenen Anzeigefarben zu den eigentlichen Farbnummerbereichen für die oben behandelten *Text-*, *Hintergrund-* und *Rahmen*-Farben flexibel zu sein. Weitere Einzelheiten finden Sie bei der Anweisung **PALETTE.**

Vergleichen Sie auch

PAINT, PALETTE, SCREEN

Beispiele

Die nachstehenden Beispielfolgen zeigen **COLOR**-Anweisungen und ihre Wirkungen in den verschiedenen Bildschirmmodi:

```
SCREEN 0
'Textfarbe=1, Hintergrund=2, Rahmen=3
COLOR 1, 2, 3
SCREEN 1
'Hintergrund=1, gerade Paletten-Nummer
COLOR 1,0
'Hintergrund=2, ungerade Paletten-Nummer
COLOR 2,1
SCREEN 7
'Textfarbe=3, Hintergrund=5
COLOR 3,5
```

COM-Anweisung

Funktion

Schaltet die Ereignisverfolgung der Datenübertragungsaktivitäten an dem angegebenen Anschluß ein oder aus oder sperrt sie.

Syntax

COM(*n*) ON

COM(*n*) OFF

COM(*n*) STOP

Anmerkungen

Der Parameter *n* ist die Nummer der Datenübertragungs-Schnittstelle; *n* kann entweder 1 oder 2 sein.

Die Anweisung **COM ON** schaltet die Datenübertragungs-Ereignisverfolgung ein. Wenn nach der Anweisung **COM ON** ein Zeichen an der Datenübertragungs-Schnittstelle ankommt, wird die in der **ON COM**-Anweisung angegebene Unterroutine aufgerufen.

COM OFF schaltet die Datenübertragungs-Ereignisverfolgung aus. Solange keine weitere **COM ON**-Anweisung ausgeführt wird, findet keine Übertragungs-Verfolgung statt. Während der ausgeschalteten Verfolgung stattfindende Ereignisse werden ignoriert.

COM STOP sperrt die Datenübertragungs-Ereignisverfolgung, so daß bis zur Ausführung einer **COM ON**-Anweisung keine Verfolgung stattfindet. Während der ausgeschalteten Verfolgung vorkommende Ereignisse werden gespeichert und nach Ausführung der nächsten **COM ON**-Anweisung bearbeitet.

Weitere Informationen zu dem Thema Ereignisverfolgung finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung", in *Programmieren in BASIC: Ausgewählte Themen.*

Vergleichen Sie auch

ON *Ereignis*

Beispiel

Eine Übersicht, wie Sie eine Ereignisverfolgung ausführen, finden Sie in den Beispielen zur Ereignisverfolgung in Kapitel 6, "Fehler- und Ereignisverfolgung", in *Programmieren in BASIC: Ausgewählte Themen.*

COMMAND$-Funktion

Funktion

Gibt die Befehlszeile an, die zum Aufrufen des Programms benutzt wurde.

Syntax

COMMAND$

Anmerkungen

Die **COMMAND$**-Funktion gibt die vollständige Befehlszeile einschließlich aller wahlfreien Parameter an, die nach ihrem QuickBASIC-Programmnamen eingegeben wurden. **COMMAND$** entfernt alle führenden Leerzeichen aus der Befehlszeile und wandelt alle Buchstaben in Großbuchstaben um. **COMMAND$** kann in selbständig ausführbaren Dateien benutzt werden oder, falls Sie in der QuickBASIC-Umgebung ausführen, durch Verwendung der Option **/cmd** in der Befehlszeile oder durch Auswahl von "**Ändere COMMAND$**" im Menü "Ausführen".

Beispiel

Das Unterprogramm `BefZeile` im nachstehenden Beispiel teilt die Befehlszeile in einzelne Argumente auf und speichert diese in einem Datenfeld. Jedes Argument wird von den nebenstehenden Argumenten durch ein oder mehrere Leerzeichen in der Befehlszeile getrennt.

```
'In diesem Modul ist Ganzzahl der Standard-
'Variablentyp.
DEFINT A-Z
'Deklariere sowohl das Unterprogramm BefZeile als auch
'die Anzahl und Typen seiner Parameter.
DECLARE SUB BefZeile(N, A$(),Max)
DIM A$(1 TO 15)
'Lies, was auf der Befehlszeile eingegeben wurde.
CALL BefZeile(N,A$(),10)
'Zeige jeden Teil der Befehlszeile an.
PRINT "Anzahl der Argumente = ";N
PRINT "Argumente sind: "
FOR I=1 TO N : PRINT A$(I) : NEXT I
```

```
'Unterroutine zum Lesen der Befehlszeile
'und zum Aufteilen der Befehlszeile in Argumente.
'Parameter:    AnzArg: Anzahl der auf der Befehlszeile
'                      gefundenen Argumente.
'              Arg$(): Datenfeld, das die Argumente
'                      aufnimmt.
'              MaxArg: Maximale Anzahl der Argumente,
'                      die das Datenfeld angeben kann.

SUB BefZeile (AnzArg, Arg$(1), MaxArg)    STATIC
CONST TRUE=-1, FALSE=0

   AnzArg=0 : In=FALSE
'Lies die Befehlszeile mit der Funktion COMMAND$.
   Cl$=COMMAND$
   L=LEN(Cl$)
'Durchlaufe die Befehlszeile Zeichen für Zeichen.
   FOR I=1 TO L
      C$=MID$(Cl$,I,1)
   'Überprüfe, ob Zeichen Leerzeichen oder Tab ist.
      IF (C$<>" " AND C$<>CHR$(9)) THEN
   'Weder Leerzeichen noch Tab.
   'Prüfung um festzustellen, ob Sie sich bereits in
   'einem Argument befinden.
         IF NOT In THEN
      'Der Beginn eines neuen Argumentes ist gefunden.
      'Prüfung auf zuviele Argumente.
            IF AnzArg=MaxArg THEN EXIT FOR
            AnzArg=AnzArg+1
            In=TRUE
         END IF
      'Füge das Zeichen dem aktuellen Argument hinzu.
         Arg$(AnzArg)=Arg$(AnzArg)+C$
      ELSE
   'Leerzeichen oder Tab gefunden.
   'Setze Flagge "Nicht in einem Argument" auf FALSE.
         In=FALSE
      END IF
   NEXT I

END SUB
```

Beispielhafte Befehlszeile und Ausgabe für eine selbständig lauffähige Datei (der angenommene Programmname ist `arg.exe`):

```
Arg eins   zwei  drei  vier   fünf   sechs
```

Ausgabe

```
Anzahl der Argumente =  6
Argumente sind:
EINS
ZWEI
DREI
VIER
FÜNF
SECHS
```

COMMON-Anweisung

Funktion

Definiert globale Variablen, die zwischen Modulen geteilt oder mit anderen Programmen verkettet werden.

Syntax

COMMON [**SHARED**][*/Blockname/*]*Variablenliste*

Anmerkungen

Die folgende Liste beschreibt die Teile der **COMMON**-Anweisung:

Teil	*Beschreibung*
SHARED	Ein wahlfreies Attribut, das anzeigt, daß die Variablen mit allen **SUB**- oder **FUNCTION**-Prozeduren in dem Modul geteilt werden. **SHARED** kann die Angabe einer **SHARED**-Anweisung oder eine andere **COMMON**-Anweisung innerhalb einer **SUB**- oder **FUNCTION**-Prozedur überflüssig machen.
Blockname	Ein gültiges BASIC-Kennzeichen (mit bis zu 40 Zeichen), das dazu benutzt wird, verschiedene Variablengruppen zu kennzeichnen. Benutzen Sie einen *Blocknamen,* um nur bestimmte Gruppen von Variablen zu teilen. Wenn ein *Blockname* vergeben wird, ist es ein benannter **COMMON**-Block. Wenn *Blockname* nicht vergeben ist, ist der Block ein unbenannter **COMMON**-Block. Größen in einem benannten **COMMON**-Block bleiben bei der Verkettung zu einem neuen Programm nicht erhalten. Siehe "Wie Sie benannte **COMMON**-Blöcke benutzen" und "Wie Sie **COMMON** mit CHAIN benutzen" auf den folgenden Seiten.

Teil	***Beschreibung***
Variablenliste	Eine Liste von Variablen und Datenfeldern, die von Modulen oder verketteten Programmen benutzt werden. Jede Variable darf nur in einer **COMMON**-Anweisung erscheinen .

Eine *Variablenliste* hat folgende Syntax :

Variable [( )]**AS** *Typ*] [, *Variable* [( )][**AS** *Typ*]]...

Die folgende Liste beschreibt die Teile einer *Variablenliste*:

Teil	***Beschreibung***
Variable	Ein beliebiger gültiger BASIC-Variablenname.
AS *Typ*	Deklariert den Typ von *Variablen* als *Typ*. Der Typ kann hierbei **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder ein vom Benutzer definierter Typ sein.

Hinweis Ältere BASIC-Versionen erfordern die Anzahl der Dimensionen, die nach dem Namen eines dynamischen Datenfeldes in einer **COMMON**-Anweisung erscheinen müssen. Die Anzahl der Dimensionen ist nicht länger erforderlich, obgleich QuickBASIC die ältere Syntax akzeptiert, um die Kompatibilität zu früheren Versionen zu wahren.

Eine **COMMON**-Anweisung reserviert Speicherplatz für Variablen in einem besonderen Bereich, damit auf sie von Modulen oder anderen Programmen zugegriffen werden kann, die mit einer **CHAIN**-Anweisung aufgerufen werden.

Da **COMMON**-Anweisungen globale Variablen für ein ganzes Programm festlegen, müssen diese Anweisungen vor der ersten ausführbaren Anweisung erscheinen. Alle Anweisungen, bis auf die folgenden, sind ausführbar:

- **COMMON**
- **CONST**
- **DATA**
- **DECLARE**
- **DEF***Typ*
- **DIM** (für statische Datenfelder)
- **OPTION BASE**
- **REM**
- **SHARED**
- **STATIC**
- **TYPE...END TYPE**
- Alle Metabefehle

Variablen in **COMMON**-Anweisungen werden nicht durch ihre Namen sondern durch ihre Positionen und ihre Typen miteinander verglichen. Daraus folgt, daß in **COMMON**-Anweisungen die Variablenanordnung wichtig ist. In dem folgenden Ausschnitt ist es nicht der Variablenname, sondern die Variablenanordnung, die die Variablen miteinander verbindet:

```
'Hauptprogramm.
COMMON A, D, E
A = 5 : D = 8 : E = 10
.
.
.
'Common-Anweisung in einem anderen Modul.
COMMON A, E, D        'A = 5, E = 8, D = 10
.
.
.
```

Beide, statische und dynamische, Datenfelder werden in **COMMON** durch Verwendung des Datenfeldnamens, gefolgt von Klammern, gesetzt. Ein statisches Datenfeld muß in einer der **COMMON**-Anweisung vorhergehenden **DIM**-Anweisung mit ganzzahligen Konstanten als Indizes dimensioniert worden sein.

Wie Datenfelder in einer **COMMON**-Anweisung deklariert und gespeichert werden, hängt davon ab, ob es sich um dynamische oder statische Datenfelder handelt. Ein dynamisches Datenfeld muß in einer späteren **DIM**- oder **REDIM**-Anweisung dimensioniert werden. Die Elemente eines dynamischen Datenfeldes werden nicht dem **COMMON**-Block zugeordnet. Es wird nur ein Datenfeld-Beschreiber in **COMMON** plaziert. (In Kapitel 2, "Datentypen", und Anhang C, "Metabefehle", finden Sie weitere Informationen zum Thema statische und dynamische Datenfelder.)

Die Größe eines **COMMON**-Bereiches kann in einem anderen Modul oder durch ein verkettetes Programm verändert werden, wenn unbenanntes **COMMON** benutzt wurde. Wenn ein BASIC-Programm **COMMON**-Blöcke mit einer Routine in der Benutzerbibliothek teilt, darf das aufrufende Programm den **COMMON**-Block nicht vergrößern.

Fehler, die durch Unverträglichkeiten in **COMMON**-Anweisungen verursacht werden, sind verzwickt und schwierig zu finden. Ein einfacher Weg, solche unverträglichen **COMMON**-Anweisungen zu vermeiden, besteht darin, die **COMMON**-Deklarationen in einer einzigen "Include"-Datei zu plazieren und den **$INCLUDE**-Metabefehl in jedem Programm zu verwenden. In Anhang C, "Metabefehle", finden Sie weitere Informationen zum Thema **$INCLUDE**.

Im folgenden Programmausschnitt wird gezeigt, wie der **$INCLUDE**-Metabefehl benutzt wird, um eine Datei, die **COMMON**-Anweisungen enthält, unter Programmen zu teilen:

```
'Diese Datei ist menue.bas.
'$INCLUDE:'COMDEF.BI'
.
.
.
CHAIN "PROG1"
END
'Diese Datei ist Prog1.bas.
'$INCLUDE:'COMDEF.BI'
.
.
.
END
'Diese Datei ist comdef.bi.
DIM A(100),B$(200)
COMMON I,J,K,A()
COMMON A$,B$(),X,Y,Z
'Ende comdef.bi.
```

Die nächsten drei Abschnitte behandeln die Verwendung von benannten **COMMON**-Blöcken unter Verwendung des **SHARED**-Schlüsselwortes und von **COMMON** für das Verketten von Programmen.

Wie Sie benannte COMMON-Blöcke benutzen

Ein benannter **COMMON**-Block bietet eine einfache Möglichkeit, Variablen so anzuordnen, daß verschiedene Module nur zu den Variablen Zugang haben, die sie benötigen.

Das folgende Programm berechnet den Inhalt und die Dichte eines rechteckigen Prismas. Dazu benutzt es einen benannten **COMMON**-Block, um verschiedene Datensätze mit zwei Unterprogrammen zu teilen. Das Unterprogramm `INHALT` muß nur die Variablen kennen, die die Länge der Seiten repräsentieren (im **COMMON**-Block `SEITEN`). Das Unterprogramm `DICHTE` benötigt die Variablen, die das Gewicht enthalten (im **COMMON**-Block `GEWICHT`).

```
'Hauptprogramm
DIM S(3)
COMMON /Seiten/ S()
COMMON /Gewicht/ C
C=52
S(1)=3: S(2)=3: S(3)=6
CALL Inhalt
CALL Dichte
END
'Unterprogramm INHALT in einem separaten Modul.
DIM S(3)
COMMON SHARED /Seiten/ S()
SUB Inhalt STATIC
   Inh=S(1) * S(2) * S(3)
   PRINT "Der Inhalt ist " Inh
END SUB
'Unterprogramm DICHTE in einem separaten Modul.
DIM S(3)
COMMON SHARED /Seiten/ S()
COMMON SHARED /Gewicht/ W
SUB Dichte STATIC
   Inh=S(1) * S(2) * S(3)
   Dch=W/Inh
   PRINT "Die Dichte ist " Dch
END SUB
```

Hinweis Benannte **COMMON**-Blöcke bleiben bei der Verkettung von Programmen nicht erhalten. Um Variablen an verkettete Programme zu übergeben, müssen Sie unbenannte **COMMON**-Blöcke verwenden.

Wie Sie COMMON mit CHAIN benutzen

Die **COMMON**-Anweisung stellt die einzige Möglichkeit dar, Variablenwerte direkt an verkettete Programme zu übergeben. Um Variablen zu übergeben, müssen beide Programme **COMMON**-Blöcke enthalten. Erinnern Sie sich, daß nicht die Namen, sondern der Typ und die Anordnung der Variablen wichtig sind. Die Anordnung und die Typen der Variablen müssen für alle **COMMON**-Anweisungen, die zwischen verkettenden und verketteten Programmen vermitteln, gleich sein.

Obwohl die Anordnung und der Typ der Variablen bestimmend für die Übergabe der richtigen Werte sind, müssen die **COMMON**-Blöcke nicht die gleiche Größe haben. Wenn die Größe des **COMMON**-Blockes im verketteten Programm kleiner als die des verkettenden Programms ist, werden die zusätzlichen **COMMON**-Variablen in den verkettenden Programmen ignoriert. Wenn der **COMMON**-Block des verketteten Programmes größer ist, werden die zusätzlichen **COMMON**-Zahlenvariablen auf Null initialisiert, die zusätzlichen Zeichenkettenvariablen auf Null-Zeichenketten.

Statische Datenfelder, die vom verkettenden Programm mit **COMMON** übergeben werden, müssen auch im verketteten Programm als statisch deklariert werden. Ähnlich müssen dynamische Datenfelder des verkettenden Programmes im verketteten Programm ebenfalls als dynamisch deklariert sein.

Hinweis Um **COMMON** mit **CHAIN** beim Kompilieren außerhalb der QuickBASIC-Umgebung zu nutzen, müssen Sie das **BRUN40.EXE**-Modul benutzen. Dieses Modul wird aufgerufen, wenn Sie Ihr Programm von der Befehlszeile ohne die **/o**-Option kompilieren, oder wenn Sie die Option "EXE erfordert BRUN40.EXE" von dem Dialogfeld **EXE-Datei erstellen** wählen.

Vergleichen Sie auch

CALL (BASIC), CHAIN, FUNCTION, SUB

Beispiel

Die folgenden beiden Beispiele benutzen eine **COMMON**-Anweisung für die Variablenübergabe zwischen zwei verketteten Programmen. Das erste Programm liest eine Serie von numerischen Werten, speichert die Werte in einem Datenfeld und verkettet dann zum anderen Programm. Das zweite Programm errechnet den Durchschnitt und gibt das Ergebnis aus.

```
DIM Werte(1 TO 50)
COMMON Werte(), AnzWerte

PRINT "Einen Wert pro Zeile eingeben."
PRINT "END eingeben zum Beenden.
AnzWerte=0
DO
   INPUT "-> ",N$
   IF I>=50 OR UCASE$(N$)="END" THEN EXIT DO
   AnzWerte=AnzWerte+1
   Werte(AnzWerte)=VAL(N$)
LOOP

CHAIN "DSchnitt"
```

Dieses mit `DSchnitt` benannte Programm sieht wie folgt aus (beachten Sie, daß `DSchnitt` ein vollständig separates Programm ist):

```
DIM X(1 TO 50)
COMMON X(), N
IF N>0 THEN
   Sum=0
   FOR I=1 TO N
      Sum=Sum+X(I)
   NEXT I
   PRINT "Der Durchschnitt der Werte ist";Sum/N
END IF
```

CONST-Anweisung

Funktion

Deklariert symbolische Konstanten, die an Stelle von Zahlen- oder Zeichenketten-Werten benutzt werden können.

Syntax

CONST *Konstantenname* = *Ausdruck* [, *Konstantenname* = *Ausdruck*]...

Anmerkungen

Argument	***Beschreibung***
Konstantenname	Ein Name, der denselben Regeln unterliegt, wie ein BASIC-Variablenname. Sie können dem Namen ein Typzeichen (%, &, #, ! oder $) hinzufügen, um seinen Typ festzulegen; dieses Zeichen ist jedoch nicht Teil des Namens.
Ausdruck	Ein Ausdruck bestehend aus Zeichen (wie `1.0`), anderen Konstanten, beliebigen arithmetischen und logischen Operatoren, außer dem Potenz-Operator (^); oder eine Zeichenkette wie `Eingabefehler`. Eingebaute Funktionen, wie **SIN** oder **CHR$**, Variablen oder benutzerdefinierte Funktionen, dürfen nicht in Konstantenzuweisungen erscheinen.

Sie können das Typzeichen weglassen, wenn Sie die Konstante, wie im folgenden Beispiel gezeigt, benutzen:

```
CONST MAXDIM% = 250
.
.
.
DIM KontoNamen$(MAXDIM)
```

Wenn Sie das Typzeichen weglassen, bekommt die Konstante den sich aus dem Ausdruck in der **CONST**-Anweisung ergebenden Typ. Zeichenketten definieren immer Zeichenketten-Konstanten. Numerische Ausdrücke werden berechnet, und der Konstanten wird der einfachste Typ, mit dem die Konstante dargestellt werden kann, zugewiesen. Wenn z. B. der Ausdruck ein Ganzzahl-Ergebnis hat, bekommt die Konstante einen ganzzahligen Typ.

Hinweis Namen von Konstanten werden durch **DEF**-Anweisungen wie **DEFINT** nicht beeinflußt. Der Typ einer Konstanten ist entweder explizit durch ein Typzeichen oder durch den Typ des Ausdrucks festgelegt.

Konstanten müssen vor ihrem Aufruf definiert sein. Das folgende Beispiel ist fehlerhaft, weil die Konstante `EINS` nicht definiert ist, bevor sie zur Berechnung von `ZWEI` benutzt wird.

```
CONST ZWEI = EINS + EINS; EINS = 1
```

Konstanten, die in einer **SUB** oder **FUNCTION** deklariert werden, sind lokal zu der **SUB** oder **FUNCTION**. Eine Konstante, die außerhalb einer Prozedur deklariert wird, ist im ganzen Modul definiert. Sie können Konstanten überall da verwenden, wo Sie auch Ausdrücke verwenden könnten.

Eine gängige Programmierpraxis ist es, eine Anweisung wie die folgende zu benutzen (beachten Sie, daß ein Wert ungleich Null den Wert "wahr" darstellt):

```
WAHR=-1
```

Konstanten bieten einige Vorteile zu Variablen, die für konstante Werte benutzt werden:

1. Konstanten können nur einmal für das gesamte Modul definiert werden.
2. Konstanten können nicht aus Versehen geändert werden.
3. In selbständig ausführbaren Programmen führt die Benutzung von Konstanten zu einem effizienteren Code als für Variablen.
4. Konstanten erleichtern die Modifizierung von Programmen.

Beispiele

Das folgende Programm benutzt die **CONST**-Anweisung, um symbolische Konstanten für die ASCII-Werte von nicht druckbaren Zeichen, wie z. B. Tabulator und Zeilenvorschub, zu deklarieren.

```
'Dieses Programm zählt Wörter, Zeilen und Zeichen.
'Ein Wort ist jede Folge von Zeichen, die keine
'Leerzeichen sind.
DEFINT a-z
CONST BLANK = 32, ENDDATEI = 26, CR = 13, LF = 10
CONST TABC = 9, YES = -1, NO = 0
'Lies den Dateinamen von der Befehlszeile.
DateiName$=COMMAND$
IF DateiName$="" THEN
   PRINT "Geben Sie den Namen der Eingabedatei ein";
   INPUT ":", DateiName$
   IF DateiName$="" THEN END
END IF
OPEN DateiName$ FOR INPUT AS #1
Woerter=0
Zeilen=0
Zeichen=0
'Setze eine Flagge um anzuzeigen, daß nicht ein Wort
'betrachtet wird, lies dann das erste Zeichen aus
'der Datei.
InWort=NO
DO UNTIL EOF(1)
   C=ASC(INPUT$(1,#1))
   Zeichen=Zeichen+1
   IF C=BLANK OR C=CR OR C=LF OR C=TABC THEN
'Wenn das Zeichen ein Leerzeichen, Wagenrücklauf,
'Zeilenvorschub oder Tab ist, befinden Sie sich nicht
'in einem Wort.
         IF C=CR THEN Zeilen=Zeilen+1
         InWort=NO
    ELSEIF InWort=NO THEN
'Das Zeichen ist ein druckbares Zeichen, es ist also
'der Anfang eines Wortes.
'Zähle das Wort und setze die Flagge
       InWort=YES
       Woerter=Woerter+1
   END IF
```

```
LOOP
PRINT Zeichen, Woerter, Zeilen
END
```

Mit Hilfe von Konstanten können Sie Ihr Programm einfacher verändern. Der folgende Programmausschnitt deklariert eine einzige Konstante, um mehrere Datenfelder zu dimensionieren. Um die Größe der Datenfelder zu verändern, ist es lediglich nötig, den Wert in der **CONST**-Anweisung zu ändern.

```
CONST MAXKUNDEN = 250
.
.
.
DIM KontoNummer$(MAXKUNDEN), Bilanz(MAXKUNDEN)
DIM Kontakt$(MAXKUNDEN), SchuldBetr(MAXKUNDEN)
.
.
.
```

Hinweis Wenn Sie hier eine Variable an Stelle einer symbolischen Konstante benutzen, wird das Datenfeld ein dynamisches Datenfeld. Das Benutzen einer symbolischen Konstante, wie im vorhergehenden Beispiel, deklariert diese Datenfelder zu statischen Datenfeldern. In Abschnitt 2, "Datentypen", und Anhang C, "Metabefehle", finden Sie weitere Informationen zu dynamischen und statischen Datenfeldern.

COS-Funktion

Funktion

Gibt den Kosinus des im Bogenmaß gegebenen Winkels an.

Syntax

COS (*Numerischer Ausdruck*)

Anmerkungen

Der Ausdruck *Numerischer Ausdruck* kann von beliebigem numerischen Typ sein.

Die Berechnung des Kosinus ist voreingestellt mit einfacher Genauigkeit. Wenn *Numerischer Ausdruck* ein Wert doppelter Genauigkeit ist, wird der Kosinus mit doppelter Genauigkeit berechnet.

Sie können eine Winkelmessung von Grad in Bogenmaß umrechnen, indem Sie die Gradwerte mit π /180 multiplizieren, wobei π = 3,141593.

Vergleichen Sie auch

ATN, SIN, TAN

Beispiel

Das nachstehende Programm zeichnet die Kurve der Polargleichung $r = n\theta$ für n-Werte von 0,1 - 1,1. Dieses Programm benutzt die Umrechnungsfaktoren $x = \cos(\theta)$ und $y = \sin(\theta)$ zur Änderung der Polarkoordinaten in kartesische x,y-Koordinaten. Die dargestellte Figur wird auch die "Spirale des Archimedes" genannt.

```
CONST PI = 3.141593
'Grauer Hintergrund.
SCREEN 1 : COLOR 7
'Definiere Fenster groß genug für größte Spirale.
WINDOW (-4,-6)-(8,2)
'Zeichne Gerade vom Ursprung nach rechts.
LINE (0,0)-(2.2*PI,0),1
'Zeichne zehn Spiralen.
FOR N = 1.1 to .1 STEP -.1
   'Zeichne Startpunkt.
   PSET (0,0)
   FOR Winkel = 0 TO 2*PI STEP .04
       'Polargleichung für Spirale.
       R = N * Winkel
       'Wandle Polarkoordinaten in kartesische
       'Koordinaten um.
       X = R * COS(Winkel)
       Y = R * SIN(Winkel)
       'Zeichne Gerade vom vorhergehenden zum neuen
       'Punkt.
       LINE -(X,Y),1
    NEXT
NEXT
```

CSNG-Funktion

Funktion

Wandelt den numerischen Ausdruck in einen Wert einfacher Genauigkeit um.

Syntax

CSNG (*Numerischer Ausdruck*)

Anmerkungen

Die Funktion **CSNG** hat dieselbe Wirkung wie die Zuweisung von *Numerischer Ausdruck* zu einer Variablen einfacher Genauigkeit.

Wenn notwendig, rundet **CSNG** den Wert vor der Umwandlung.

Vergleichen Sie auch

CDBL, CINT

Beispiele

Das folgende Beispiel zeigt, wie **CSNG** den Wert vor der Umwandlung rundet.

```
A#=975.34#
B#=975.35#
PRINT A#; CSNG(A#); B#; CSNG(B#)
```

Ausgabe

```
 975.3421115   975.3421   975.3421555   975.3422
```

CSRLIN-Funktion

Funktion

Ermittelt die aktuelle Zeilenposition des Cursors.

Syntax

CSRLIN

Anmerkungen

Zur Ermittlung der aktuellen Spaltenposition verwenden Sie die Funktion **POS**.

Vergleichen Sie auch

LOCATE, POS

Beispiel

Das folgende Programm ruft ein Unterprogramm auf, das ohne Änderung der aktuellen Cursorposition eine Meldung auf den Bildschirm ausgibt.

```
'Bewege Cursor zur Mitte des Bildschirms, gib dann
'eine Meldung aus.
'Cursor in Bildschirmmitte positionieren.
   LOCATE 12,40
   CALL MldOhneBew ("Feierabend!",9,35)
   INPUT "Beenden mit jeder Eingabe: ",A$
'Gib eine Meldung aus, ohne die aktuelle
'Cursorposition zu zerstören.
SUB MldOhneBew(Nachr$,Zeile%,Spalte%) STATIC
   'Sichere die aktuelle Zeile.
   AktZeile%=CSRLIN
   'Sichere die aktuelle Spalte.
   AktSpalte%=POS(0)
   'Gib die Meldung an der gegebenen Position aus.
   LOCATE Zeile%,Spalte% : PRINT Nachr$;
   'Bewege den Cursor zurück zur ursprünglichen
   'Position.
   LOCATE AktZeile%, AktSpalte%
END SUB
```

CVI-, CVS-, CVL-, CVD-Funktionen

Funktion

Wandelt Zeichenketten, die numerische Werte enthalten, in Zahlen um.

Syntax

CVI (*2-Byte-Zeichenkette*)
CVS (*4-Byte-Zeichenkette*)
CVL (*4-Byte-Zeichenkette*)
CVD (*8-Byte-Zeichenkette*)

Anmerkungen

CVI, **CVS**, **CVL** und **CVD** werden zusammen mit der Anweisung **FIELD** benutzt, um reelle Zahlen von einer Datei mit wahlfreiem Zugriff zu lesen. Die Funktionen nehmen in der **FIELD**-Anweisung definierte Zeichenketten und konvertieren diese in Werte des zugehörigen numerischen Typs. Diese Funktionen sind das Gegenstück zu **MKI$**, **MKS$**, **MKL$** und **MKD$**:

Funktion	*Beschreibung*
CVI	Wandelt eine mit **MKI$** erstellte *2-Byte-Zeichenkette* in eine ganze Zahl zurück.
CVS	Wandelt eine mit **MKS$** erstellte *4-Byte-Zeichenkette* in eine Zahl einfacher Genauigkeit zurück.
CVL	Wandelt eine mit **MKL$** erstellte *4-Byte-Zeichenkette* in eine 4-Byte-Ganzzahl zurück.
CVD	Wandelt eine mit **MKD$** erstellte *8-Byte-Zeichenkette* in eine Zahl doppelter Genauigkeit zurück.

Hinweis Die neuen BASIC-Verbundvariablen ermöglichen dem Benutzer effizienteres und einfacheres Lesen und Schreiben von Direktzugriffsdateien. In Kapitel 3, "Datei- und Geräte E/A", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie ein komplettes Beispiel.

Vergleichen Sie auch

FIELD, **LSET**, **MKD$**, **MKI$**, **MKL$**, **MKS$**

Beispiel

Das folgende Beispiel illustriert den Gebrauch von **MKS$** und **CVS**:

```
OPEN "KONTO.INF" FOR RANDOM AS #2 LEN = 29
FIELD #2, 25 AS Name$, 4 AS Scheck$
Format$ = "$$#####.##"
DO
   PRINT
   DO
      INPUT "Geben Sie eine Konto-Nr. zum _
             Aktualisieren ein: ",Satz%
      GET #2, Satz%  'Hole den Satz
      PRINT "Dies ist das Konto für " Name$
      INPUT "Ist dies das gewünschte Konto"; R$
   LOOP WHILE UCASE$(MID$(R$,1,1)) <> "J"
   'Wandelt Zeichenkette in Zahl einfacher Genauigkeit
   'um.
   Scheckbetr! = CVS(Scheck$)
   PRINT
   PRINT "Die Eröffnungsbilanz für dieses Konto ist";
   PRINT USING Format$; Scheckbetr!
   PRINT "Geben Sie Schecks und Barabhebungen für"
   PRINT "dieses Konto ein. Geben Sie 0 ein zum ";
   PRINT "Beenden."
   DO
      INPUT Scheckaus!
      Scheckbetr! = Scheckbetr! - Scheckaus!
   LOOP UNTIL Scheckaus! = 0
   PRINT
   PRINT "Geben Sie die Einzahlungen für dieses Konto"
   PRINT "ein. Geben Sie 0 ein zum Beenden."
   DO
      INPUT Scheckein!
      Scheckbetr! = Scheckbetr! + Scheckein!
   LOOP UNTIL Scheckein! = 0
   PRINT
   PRINT "Die Schlußbilanz für dieses Konto ist";
   PRINT USING Format$; Scheckbetr!
   'Wandle Zahl einfacher Genauigkeit in Zeichenkette
   'um.
   LSET Scheck$ = MKS$(Scheckbetr!)
   PUT #2, Satz%  'Speichere den Satz.
   INPUT "Ein anderes Konto aktualisieren"; R$
```

```
LOOP UNTIL UCASE$(MID$(R$,1,1)) <> "J"
CLOSE #2
END
```

CVSMBF-, CVDMBF-Funktionen

Funktion

Überträgt Zeichenketten, die Zahlen im Microsoft-Binär-Format enthalten, in Zahlen mit IEEE-Format.

Syntax

CVSMBF (*4-Byte-Zeichenkette*)

CVDMBF (*8-Byte-Zeichenkette*)

Anmerkungen

CVSMBF und **CVDMBF** helfen Ihnen, alte Direktzugriffsdateien zu lesen, die als Zeichenketten im Microsoft-Binär-Format abgespeicherte Zahlen enthalten. Diese Funktionen konvertieren die Zeichenketten, die die Nummern enthalten, nachdem sie aus der alten Datei gelesen wurden, in Zahlen mit IEEE-Format:

Funktion	*Beschreibung*
CVSMBF	Konvertiert eine *4-Byte-Zeichenkette*, die eine Zahl im Microsoft-Binär-Format enthält, in eine IEEE-formatierte Zahl mit einfacher Genauigkeit.
CVDMBF	Konvertiert eine *8-Byte-Zeichenkette*, die eine Zahl im Microsoft-Binär-Format enthält, in eine IEEE-formatierte Zahl mit doppelter Genauigkeit.

Das Beispiel zeigt, wie Sie Daten aus einer alten Datei mit Hilfe von **CVSMBF** und benutzerdefinierten Satztypen lesen. Im Anhang B, "Unterschiede zu früheren Versionen von QuickBASIC", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen zur Konvertierung alter Datendateien.

Vergleichen Sie auch

FIELD, MKDMBF, MKSMBF

Beispiel

Das folgende Beispiel liest Sätze aus einer Direktzugriffsdatei, die als Zeichenketten abgespeicherte reelle Zahlen im Microsoft-Binär-Format enthält. Jeder Satz enthält den Namen eines Studenten und dessen Prüfungsergebnisse.

```
'Definiere einen Benutzertyp für den Datensatz.
TYPE StudentSatz
   NameFeld AS STRING*20
   Ergebn AS STRING*4
END TYPE
'Definiere eine Variable des benutzerdefinierten Typs.
DIM Satz AS StudentSatz
'Öffne die Datei.
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN=LEN(Satz)
Max=LOF(1)/LEN(Satz)
'Lesen und Ausgeben aller Sätze.
FOR I=1 TO Max
   'Lies einen Satz in die Benutzertyp-Variable Satz.
   GET #1,I,Satz
   'Wandle das Ergebnis von einer Zeichenkette, die
   'eine Zahl im Microsoft-Binär-Format enthält, in
   'eine Zahl im IEEE-Format um.
   ErgebAus=CVSMBF(Satz.Ergebn)
   'Gib den Namen und das Ergebnis aus.
   PRINT Satz.NameFeld,ErgebAus
NEXT I
CLOSE #1
```

DATA-Anweisung

Funktion

Speichert die numerischen und Zeichenkettenkonstanten, die von den **READ**-Anweisungen des Programms benutzt werden.

Syntax

DATA *Konstante1* [,*Konstante2*]...

Anmerkungen

Die *Konstante1, Konstante2* usw. in der **DATA**-Anweisung können jede zulässige numerische oder Zeichenketten-Konstante sein.

Namen von symbolischen Konstanten (definiert in einer **CONST**-Anweisung), die in einer **DATA**-Anweisung erscheinen, werden nicht als Konstantennamen, sondern als Zeichenketten interpretiert. In dem folgenden Programm-Ausschnitt ist die zweite Datengröße zum Beispiel die Zeichenkette "PI" und nicht der Wert 3,141593:

```
CONST PI = 3.141593
.
.
.
DATA 2.20, PI, 45, 7
.
.
.
```

Eine **DATA**-Anweisung kann soviele Konstanten enthalten, wie in eine Zeile passen. Die Konstanten werden durch Kommata getrennt.

Hinweis Zeichenkettenkonstanten erfordern in **DATA**-Anweisungen nur dann doppelte Anführungsstriche, wenn sie Kommata, Semikolons oder zu berücksichtigende führende oder folgende Leerzeichen enthalten.

Nulldatenelemente dürfen in der Datenliste erscheinen:

```
DATA  1,2,,4,5
```

Wenn ein Nulldatenelement in eine numerische Variable gelesen wird, hat sie den Wert 0. Wenn ein Nulldatenelement in eine Zeichenkettenvariable gelesen wird, hat die Variable den Wert Null-Zeichenkette ("").

Sie können beliebig viele **DATA**-Anweisungen verwenden.

Wenn Sie in der QuickBASIC-Umgebung arbeiten, können **DATA**-Anweisungen nur im Modul-Ebenen-Code erscheinen. QuickBASIC verschiebt beim Lesen der Quelldatei alle **DATA**-Anweisungen, die nicht im Modul-Ebenen-Code stehen in den Modul-Ebenen-Code. **READ**-Anweisungen, die **DATA**-Anweisungen benutzen, können an jeder Stelle im Programm erscheinen.

DATA-Anweisungen werden in der Reihenfolge, in der sie in der Quelldatei erscheinen, ausgewertet. Sie können sich die Größen in verschiedenen **DATA**-Anweisungen als geschlossene Folge von Größen vorstellen, unabhängig von der Anzahl der Größen in der Anweisung oder wo die Anweisung im Programm steht.

Vergleichen Sie auch

READ, RESTORE

Beispiele

Das folgende Beispiel zeigt das von der Funktion **DATE$** gelesene und anschließend konvertierte aktuelle Datum an.

```
'Lies das Datum.
C$ = DATE$
'Verwende VAL, um den Monat aus der von DATE$
'angegebenen Zeichenkette herauszufinden.
FOR I% = 1 TO VAL(C$)
   READ Monat$
NEXT
DATA Januar, Februar, Maerz, April, Mai, Juni, Juli
DATA August, September, Oktober, November, Dezember
'Lies den Tag.
Tag$ = MID$(C$,4,2)
IF LEFT$(Tag$,1) = "0" THEN Tag$ = RIGHT$(Tag$,1)
'Lies das Jahr.
Jahr$ = RIGHT$(C$,4)
PRINT "Heute ist der " Tag$ ". " Monat$ " " Jahr$
```

Ausgabe

```
Heute ist der 26. Maerz 1987
```

Das folgende Beispiel zeigt, wie Nulldatengrößen behandelt werden:

```
DATA abc,,def
DATA 1,,2
READ A$, B$, C$          'B$ = ""
PRINT A$, B$, C$
PRINT
READ A, B, C             'B = 0
PRINT A, B, C
```

Ausgabe

```
abc                            def
 1              0               2
```

DATE$-Anweisung

Funktion

Setzt das aktuelle Datum.

Syntax

DATE$ = *Zeichenkettenausdruck*

Anmerkungen

Diese Anweisung ergänzt die Funktion **DATE$.**

Der *Zeichenkettenausdruck* muß eines der folgenden Formate haben, wobei *mm* und *dd* Monat und Tag, *jj* und *jjjj* das Jahr sind.

mm-tt-jj

mm-tt-jjjj

mm/tt/jj

mm/tt/jjjj

Beispiel

Das folgende Programm setzt das aktuelle Datum mit einem eingegebenen numerischen Datum:

```
PRINT "Geben Sie nachfolgend das Datum ein";
PRINT "(Vorgabe für Jahr: 1987)"
INPUT "   Monat:  ",Monat$
INPUT "   Datum:  ",Tag$
INPUT "   Jahr:   ",Jahr$
IF Jahr$ = "" THEN Jahr$ = "87"
DATUM$ = Monat$ + "/" + Tag$ + "/" + Jahr$
```

DATE$-Funktion

Funktion

Gibt eine Zeichenkette an, die das aktuelle Datum enthält.

Syntax

DATE$

Anmerkungen

Die Funktion **DATE$** gibt eine Zehn-Zeichen-Zeichenkette im Format *mm-tt-jjjj* an, wobei *mm* der Monat (01-12), *tt* der Tag (01-31) und *jjjj* das Jahr (1980-2099) ist.

Vergleichen Sie auch

DATE$-Anweisung

Beispiel

Beachten Sie bitte, daß die Funktion `DATE$` im nachfolgenden Beispiel vor dem Monat eine Null ausgibt.

```
PRINT DATE$
```

Ausgabe

```
05-20-1987
```

DECLARE-Anweisung (BASIC-Prozeduren)

Funktion

Deklariert den Bezug zu BASIC-Prozeduren und ruft eine Argumenttyp-Prüfung auf.

Syntax

DECLARE {**FUNCTION** | **SUB**} *Name* [([*Parameterliste*])]

Anmerkungen

Die Anweisung **DECLARE** hat folgende Argumente:

Argument	*Beschreibung*
Name	Der Name der Prozedur. Der Name ist auf 40 Zeichen begrenzt. **FUNCTION**-Namen können mit einem der Typdeklarationszeichen (%, &, !, # oder $) enden, um den Typ des zurückgegebenen Wertes anzuzeigen.
Parameterliste	Eine Liste von Parametern, die die Zahl und den Typ der beim Aufruf benutzten Argumente anzeigt. Die Syntax wird weiter unten beschrieben. Nur die Anzahl und der Typ der Argumente sind entscheidend.

Für Aufrufe innerhalb BASIC ist die **DECLARE**-Anweisung nur erforderlich, wenn Sie **SUB**-Prozeduren ohne das **CALL**-Schlüsselwort aufrufen, oder wenn Sie eine **FUNCTION** verwenden, die in einem anderen Programm definiert wurde. In Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie weitere Informationen über den Aufruf von Unterprogrammen ohne **CALL**.

Eine **DECLARE**-Anweisung veranlaßt den Compiler darüber hinaus, die Anzahl und Typen der beim Aufruf der Prozedur verwendeten Argumente zu überprüfen. Wenn Sie Ihr Programm sichern, während Sie in der Umgebung arbeiten, erzeugt QuickBASIC automatisch eine **DECLARE**-Anweisung. Die **DECLARE**-Anweisung kann nur im Modul-Ebenen-Code (nicht in einer **SUB** oder **FUNCTION**) vorkommen und betrifft das gesamte Modul.

Die *Parameterliste* dient als Prototyp für die Überprüfung von Anzahl und Typ der Argumente in den **SUB**- und **FUNCTION**-Aufrufen. Sie hat die folgende Syntax:

Variable [**AS** *Typ*] [,*Variable* [**AS** *Typ*]]...

Eine *Variable* ist ein beliebiger, gültiger BASIC-Variablenname. Wenn die Variable ein Datenfeld ist, kann ihr die Anzahl der Dimensionen in Klammern folgen:

```
DECLARE SUB SchreibeText (A(2) AS STRING)
DIM Text$(100,5)
.
.
CALL SchreibeText(Text$())
```

Die Anzahl der Dimensionen ist wahlfrei.

Der *Typ* ist entweder **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder benutzerdefiniert. Nur die Anzahl und der Typ sind von Bedeutung.

Hinweis Sie können in die **DECLARE**-Anweisungen keine Zeichenketten mit fester Länge schreiben, weil an **SUB**- oder **FUNCTION**-Prozeduren nur Zeichenketten mit variabler Länge übergeben werden können. Zeichenketten mit fester Länge können zwar in einer Argumentenliste erscheinen, werden aber vor der Übertragung in Zeichenketten mit variabler Länge umgewandelt.

Der Typ einer Variablen kann ebenfalls durch die explizite Angabe eines Typzeichens (%, &, !, # oder $) oder durch die Beibehaltung des Standardtyps angezeigt werden.

Die Form der Parameterliste bestimmt, ob eine Überprüfung der Argumente stattfindet oder nicht (siehe folgende Tabelle):

Deklaration	***Bedeutung***
`DECLARE SUB First`	Sie können die Klammern nur weglassen, wenn die **SUB** oder **FUNCTION** separat kompiliert wird. Eine Argumentenüberprüfung findet nicht statt.
`DECLARE SUB First ()`	`First` hat keine Parameter. Die Angabe von Argumenten in einem **CALL** von `First` wird als Fehler angezeigt. Eine leere Parameterliste zeigt an, daß die **SUB** oder **FUNCTION** keine Parameter hat.
`DECLARE SUB First (X AS LONG)`	`First` hat einen langen Ganzzahl-Parameter. Anzahl und Typ der Argumente in jedem **CALL** oder Aufruf werden überprüft, wenn die Parameterliste in **DECLARE** erscheint.

Vergleichen Sie auch

CALL, CALLS (Nicht-BASIC); FUNCTION; SUB

Beispiel

Im folgenden Beispiel erlaubt die Anweisung **DECLARE** den Aufruf von `Sort` ohne das **CALL**-Schlüsselwort.

```
' Erzeuge 20 Zufallszahlen, speichere sie in einem
' Datenfeld und sortiere sie. Das Sortier-
' Unterprogramm wird ohne das Schlüsselwort CALL
' aufgerufen.
DECLARE SUB Sort(A(1) AS SINGLE, N AS INTEGER)
DIM Feld1(1 TO 20)
' Erzeuge 20 Zufallszahlen.
RANDOMIZE TIMER
FOR I=1 TO 20
   Feld1(I)=RND
NEXT I
' Sortiere das Datenfeld. Sort wird ohne das
' Schlüsselwort CALL aufgerufen. Beachte, daß keine
' Klammern um die Argumente in dem Sort-Aufruf
' geschrieben wurden.
Sort Feld1(), 20%
' Drucke das sortierte Datenfeld.
FOR I=1 TO 20
   PRINT Feld1(I)
NEXT I
END
' Sortier-Unterprogramm
SUB Sort(A(1), N%) STATIC
   FOR I= 1 TO N%-1
      FOR J = I+1 TO N%
         IF A(I)>A(J) THEN SWAP A(I), A(J)
      NEXT J
   NEXT I
END SUB
```

DECLARE-Anweisung (Nicht-BASIC-Prozeduren)

Funktion

Deklariert den Aufruf für in anderen Sprachen geschriebene externe Prozeduren.

Syntax 1

DECLARE FUNCTION *Name* [**CDECL**][**ALIAS** *"Aliasname"*][([*Parameterliste*])]

Syntax 2

DECLARE SUB *Name* [**CDECL**][**ALIAS** *"Aliasname"*][([*Parameterliste*])]

Anmerkungen

Die folgende Liste beschreibt die Teile der **DECLARE**-Anweisung:

Teil	*Beschreibung*
FUNCTION	Zeigt an, daß die externe Prozedur einen Wert ausgibt und in einem Ausdruck benutzt werden kann.
SUB	Zeigt an, daß die externe Prozedur wie eine BASIC-**SUB** aufgerufen wird.
Name	Der im BASIC-Programm zum Aufruf der Prozedur verwendete Name. Sie können bis zu 40 Zeichen in einem Namen verwenden. **FUNCTION** kann explizite Typzeichen (%, &, !, # oder $) enthalten, die den Typ des von **FUNCTION** ausgegebenen Wertes anzeigen.
CDECL	Zeigt an, daß die Prozedur die Argumenten-Reihenfolge der Sprache C benutzt. **CDECL** übergibt die Argumente von rechts nach links, anders als bei der üblichen BASIC-Vereinbarung von links nach rechts. **CDECL** betrifft auch die Namen, die bei der Suche in Objekt-Dateien und Bibliotheken benutzt werden. Falls kein **ALIAS** im **DECLARE** vorhanden ist, wird der Name der Prozedur (*Name*) in Kleinbuchstaben umgewandelt, das Typdeklarationszeichen wird entfernt und ein Unterstreichungszeichen wird an den Anfang gesetzt. Dies wird der Name, nach dem in externen Dateien und Bibliotheken gesucht wird. Wenn **CDECL** mit **ALIAS** benutzt wird, wird der *Aliasname* benutzt.
ALIAS	Zeigt an, daß die Prozedur in der **.OBJ**- oder in der Bibliotheks-Datei einen anderen Namen hat.
Aliasname	Der Name, den die Prozedur in der Datei oder Bibliothek hat.

Eine *Parameterliste* hat folgende Syntax:

[{**BYVAL** | **SEG**}]*Variable* [**AS** *Typ*],[{**BYVAL** | **SEG** }] *Variable* [**AS** *Typ*]]...

Die folgende Liste beschreibt die Teile einer *Parameterliste*:

Teil	*Beschreibung*
BYVAL	Zeigt an, daß der Parameter als Wert und nicht als Referenz übergeben wird. Übergabe als Referenz ist voreingestellt. **BYVAL** kann nur mit den Parametern **INTEGER, LONG, SINGLE** und **DOUBLE** verwendet werden. Wenn **BYVAL** vor den Parametern steht, wird das aktuelle Argument vor der Übertragung in den in der **DECLARE**-Anweisung angezeigten Typ umgewandelt.
SEG	Zeigt an, daß der Parameter als Segmentadresse (Far Pointer) übergeben wird.
Variable	Ein gültiger BASIC-Variablenname. Von Bedeutung ist nur der Typ der Variablen. Wenn die Variable ein Datenfeld ist, kann ihr die Anzahl der Dimensionen in Klammern folgen (um die Kompatibilität zu alten BASIC-Versionen zu wahren): `DECLARE SUB EigenWert (A(2) AS DOUBLE)` Die Angabe der Anzahl der Dimensionen ist wahlfrei. In Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Angaben zur Übergabe von BASIC-Datenfeldern in Prozeduren, die in anderen Sprachen geschrieben sind.
AS *Typ*	Zeigt den Typ der Variablen. Das Element Typ kann **INTEGER, LONG, SINGLE, DOUBLE, STRING, ANY** oder benutzerdefiniert sein. Sie können den Typ der Variablen ebenso mit der Angabe des expliziten Typzeichens (%, &, !, # oder $) im Variablennamen oder durch Beibehaltung des Standardtyps anzeigen. Wenn Sie in anderen Sprachen geschriebene externe Prozeduren deklarieren, können Sie das **ANY**-Schlüsselwort in der **AS**-Klausel benutzen. **ANY** überschreibt die Typüberprüfung für dieses Argument. Sie können **ANY** nicht benutzen, wenn Sie Argumente als Wert übergeben.

Hinweis Wenn weder **BYVAL** noch **SEG** benutzt werden, werden die Argumente als Offsets übergeben.

Mit dieser Form der **DECLARE**-Anweisung können Sie Prozeduren, die in anderen Sprachen geschrieben sind, aufrufen. Die **DECLARE**-Anweisung veranlaßt den Compiler ebenso, Zahl und Typ der beim Aufruf der Prozedur verwendeten Argumente zu überprüfen. Eine **DECLARE**-Anweisung kann nur im Modul-Ebenen-Code vorkommen und ist für die gesamte Quelldatei wirksam.

Die Form der Parameterliste bestimmt, ob eine Überprüfung der Argumenttypen erfolgt oder nicht.

Deklaration	***Bedeutung***
`DECLARE SUB First CDECL`	Wenn keine Parameterliste vorhanden ist, wird keine Argument-Überprüfung durchgeführt.
`DECLARE SUB First CDECL ()`	`First` hat keine Argumente. Argumente in einem **CALL** von `First` werden als Fehler angezeigt. Leere Klammern zeigen an, daß die **SUB** oder **FUNCTION** keine Argumente hat.
`DECLARE SUB First CDECL (X AS LONG)`	`First` hat ein langes Ganzzahl-Argument. Wenn eine Parameterliste vorkommt, werden bei jedem Aufruf Zahl und Typ der Argumente überprüft.

Eine Prozedur, die in einer **DECLARE**-Anweisung erscheint, kann ohne das Schlüsselwort **CALL** aufgerufen werden. Weitere Informationen finden Sie in der Beschreibung der **DECLARE**-Anweisung (BASIC-Prozeduren).

Hinweis Sie können in die **DECLARE**-Anweisung keine Zeichenketten mit fester Länge schreiben, weil an **SUB**- oder **FUNCTION**-Prozeduren nur Zeichenketten mit variabler Länge übergeben werden können. Zeichenketten mit fester Länge können zwar in einer Argumentenliste erscheinen, werden aber vor der Übertragung in Zeichenketten variabler Länge umgewandelt.

Seien Sie vorsichtig, wenn Sie bei der Übergabe von Datenfeldern das Schlüsselwort **SEG** verwenden, weil BASIC Variablen im Speicher bewegen kann, bevor die aufgerufene Routine mit der Ausführung beginnt. In der Argumentenliste einer **CALL**-Anweisung kann alles Probleme bereiten, was Speicherbewegungen verursacht. Sie können Variablen unter Verwendung von **SEG** sicher übergeben, wenn die Argumentenliste der **CALL**-Anweisung nur einfache Variablen, arithmetische Ausdrücke oder Datenfelder, die ohne Verwendung eingebauter oder benutzerdefinierter Funktionen indiziert sind, enthält.

Eine Liste von Ereignissen, die die Bewegung von Variablen verursachen, und Informationen darüber, wann kurze oder lange Adressen für Variablen zu verwenden sind, finden Sie in Abschnitt 2.3.3, "Speicherzuweisung für Variablen".

Vergleichen Sie auch

CALL, CALLS (Nicht-BASIC); DECLARE (BASIC)

Beispiel

Das folgende Beispiel zeigt ein BASIC-Programm, das eine kurze C-Funktion aufruft. Das C-Programm wird separat kompiliert und in einer Quick-Bibliothek gespeichert oder explizit gebunden, um eine .EXE-Datei zu erstellen. In Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen über die Programmierung in verschiedenen Sprachen.

```
DEFINT a-z
' Die Funktion Addein benutzt die C-Argumentenübergabe
' und hat ein einzelnes ganzzahliges Argument, das
' nach Wert übergeben wird.
DECLARE FUNCTION Addein CDECL (BYVAL n AS INTEGER)
INPUT x
y=Addein(x)
PRINT "x und y sind ";x;y
END
```

```
/* C Funktion Addein. Gibt den um eins erhoehten Wert
     ihres ganzzahligen Argumentes aus. */
int far Addein(n)
int n;
{
   return(++n);
}
```

DEF FN-Anweisung

Funktion

Definiert und benennt eine Funktion.

Syntax 1

DEF FN*Name* [(*Parameterliste*)] = *Ausdruck*

Syntax 2

DEF FN*Name* [*(Parameterliste)*]
.
.
.
FN*Name* = *Ausdruck*
.
.
.
END DEF

Anmerkungen

Die Anweisung **DEF FN** hat folgende Argumente:

Argument	*Beschreibung*

Name	Ein zulässiger Variablenname, der bis zu 40 Zeichen lang sein kann. Dieser Name, kombiniert mit **FN**, ist der Name der Funktion. Der Name kann eine explizite Typdeklaration enthalten, um den Typ des zurückgegebenen Wertes anzuzeigen. Namen, die bis auf ihre Typdeklaration gleich lauten, sind verschiedene Namen. Beispielsweise sind die folgenden Namen die Namen drei verschiedener **DEF FN**-Funktionen:
	
	`FNZeikette$`
	`FNZeikette%`
	`FNZeikette#`
	
	Um einen Wert aus einer **DEF FN**-Funktion zurückzugeben, weisen Sie den Wert dem ganzen Funktionsnamen zu.
	
	`FNZeikette$ = "Keine Antwort."`

Parameterliste	Eine Liste von Variablennamen, die durch Kommata voneinander getrennt werden. Die Syntax wird unten erklärt. Wenn die Funktion aufgerufen wird, weist BASIC den Wert jedes Arguments ihrem entsprechenden Parameter zu. Funktionsargumente werden als Wert übergeben. **DEF FN**-Funktionen akzeptieren keine Datenfelder, Verbunde oder Zeichenketten mit fester Länge als Argumente.

Argument	***Beschreibung***
Ausdruck	In beiden Syntaxstrukturen wird der *Ausdruck* berechnet, und das Ergebnis ist der Funktionswert. In Syntax 1 ist der *Ausdruck* der gesamte Körper der Funktion und ist auf eine logische Zeile begrenzt. Wenn dem Namen kein Ausdruck zugewiesen wird, ist die voreingestellte Ausgabe der Werte Null für eine numerische **DEF FN**-Funktion und die Null-Zeichenkette ("") für eine **DEF FN**-Funktion.

Eine *Parameterliste* hat die folgende Syntax:

Variable [**AS** *Typ*] [, *Variable* [**AS** *Typ*]]...

Eine *Variable* ist jeder in BASIC gültige Variablenname. Der *Typ* ist entweder **INTEGER, LONG, SINGLE, DOUBLE** oder **STRING.** Sie können einen Variablentyp auch mit der Angabe eines Typzeichens (%, &, !, # oder $) im Namen der Variablen anzeigen.

Hinweis Die **FUNCTION**-Prozedur bietet eine bessere Kontrolle und größere Flexibilität als die **DEF FN**-Funktion. In der Beschreibung zu **FUNCTION** und in Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie weitere Informationen.

Sie müssen eine **DEF FN**-Funktion mit einer **DEF FN**-Anweisung definieren, bevor die Funktion benutzt wird. Der Aufruf einer **DEF FN**-Funktion bevor sie definiert wurde, führt zu einer Fehlermeldung `Funktion nicht definiert.` **DEF FN**-Funktionsdefinitionen können nicht innerhalb anderer **DEF FN**-Definitionen erscheinen. Außerdem können **DEF FN**-Funktionen nicht rekursiv sein.

Sie müssen die **EXIT DEF**-Anweisung benutzen, um eine mehrzeilige **DEF FN** vorzeitig zu verlassen.

DEF FN-Funktionen können nur in den Modulen, in denen sie definiert sind, verwendet werden. Sie können nicht von Prozeduren in anderen Modulen benutzt werden.

Eine **DEF FN**-Funktion kann Variablen mit dem Modul-Ebenen-Code teilen. Die nicht in der *Parameterliste* aufgeführten Variablen sind global, ihre Werte werden mit dem aufrufenden Programm geteilt. Um einen Variablenwert lokal zu einer Funktion zu definieren, müssen Sie ihn mit einer **STATIC**-Anweisung deklarieren. Siehe Kapitel 2, "Datentypen", für weitere Informationen über lokale und globale Datentypen.

DEF FN kann entweder numerische oder Zeichenketten-Werte zurückgeben. **DEF FN** gibt einen Zeichenkettenwert an, wenn *Name* ein Zeichenketten-Variablenname ist, und einen numerischen Wert, wenn *Name* ein numerischer Variablenname ist. Das Zuordnen einer numerischen Variablen zu einem Zeichenketten-Funktionsnamen oder das Zuordnen eines Zeichenkettenwertes zu einem numerischen Funktionsnamen führt zu der Fehlermeldung `Unverträgliche Datentypen.`

Ist die Funktion numerisch, so gibt **DEF FN***Name* einen Wert mit der vom Namen festgelegten Genauigkeit an. So hat beispielsweise der Wert von **DEF FN***Name* doppelte Genauigkeit, wenn *Name* eine Variable doppelter Genauigkeit ist, ungeachtet der Genauigkeit von *Ausdruck*.

Da BASIC für eine größere Effizienz arithmetische Ausdrücke neu ordnen kann, vermeiden Sie die Benutzung von **DEF FN**-Funktionen, die Programmvariablen in Ausdrücken, die neu geordnet werden können, ändert. Das folgende Beispiel kann unterschiedliche Ergebnisse liefern:

```
DEF FNKurz
   I=10
   FNKurz=1
END DEF
I=1 : PRINT FNKurz + I + I
```

Wenn BASIC den Ausdruck neu ordnet, so daß `FNKurz` nach der Berechnung von `(I+I)` aufgerufen wird, ist das Ergebnis 3, und nicht 21. Sie können dieses Problem normalerweise durch Isolierung des **DEF FN**-Funktionsaufrufes vermeiden:

```
I = 1 : X = FNKurz : PRINT X + I + I
```

Die Durchführung von E/A-Operationen in **DEF FN**-Funktionen in E/A-Anweisungen oder von grafischen Operationen in **DEF FN**-Funktionen in grafischen Anweisungen können ähnlich unvorhersehbare Ergebnisse hervorrufen.

Vergleichen Sie auch

EXIT, FUNCTION, STATIC

Beispiel

Das folgende Beispiel benutzt eine **DEF FN**-Funktion, um die Fakultät einer eingegebenen Zahl zu berechnen (die Fakultät von 3 ist 3*2*1).

```
DEF FNFakultaet#(X)
   STATIC Tmp#, I
   Tmp#=1
   FOR I=2 TO X
      Tmp#=Tmp#*I
   NEXT I
   FNFakultaet#=Tmp#
END DEF
INPUT "Bitte eine Zahl eingeben: ",Num
PRINT "Fakultät von " Num "ist" FNFakultaet#(Num)
```

Ausgabe

```
Bitte eine Zahl eingeben: 3
Fakultät von 3 ist 6
```

DEF SEG-Anweisung

Funktion

Setzt die aktuelle Segmentadresse für eine folgende **PEEK**-Funktion oder **BLOAD-**, **BSAVE-**, **CALL ABSOLUTE-** oder **POKE**-Anweisung.

Syntax

DEF SEG [= *Adresse*]

Anmerkungen

Die *Adresse* wird als das Segment für **BLOAD**, **BSAVE**, **CALL ABSOLUTE**, **PEEK** und **POKE** benutzt. Die *Adresse* ist ein numerischer Ausdruck, der eine vorzeichenlose ganze Zahl im Bereich von 0 bis 65.535 angibt. Ein Wert außerhalb dieses Bereiches produziert eine Fehlermeldung `Unzulässiger Funktionsaufruf`. Das vorhergehende Segment wird festgehalten, wenn ein Fehler erscheint.

Wenn Sie die Option *Adresse* nicht angeben, wird das BASIC-Datensegment verwendet.

Stellen Sie sicher, daß **DEF** und **SEG** durch ein Leerzeichen getrennt sind. Anderenfalls interpretiert BASIC die Anweisung als "Weise der Variablen DEFSEG einen Wert zu".

Unterschiede zu BASICA

In QuickBASIC benutzen die Anweisungen **CALL** und **CALLS** nicht die von **DEF SEG** gesetzten Segmentadressen.

Beispiel

Das folgende Programm verwendet **DEF SEG**, **PEEK** und **POKE**, um die UMSCHALT-FESTSTELLTASTE (CAPS LOCK) ein- und auszuschalten.

Hinweis Dieses Programm beinhaltet Hardware-spezifische Instruktionen. Es arbeitet korrekt auf IBM PC-, XT- und AT-Computern.

```
DECLARE SUB CapsEin ()
DECLARE SUB CapsAus ()
DECLARE SUB DruckeMeldung (R%,C%,M$)
CLS
' Schalte Caps Lock ein.
CapsEin
DruckeMeldung 24,1,"<Caps Lock ein>"
' Eingabeaufforderung, um zu zeigen, daß der
' Tastaturmodus gewechselt hat.
LOCATE 12,10
PRINT "Geben Sie eine Zeichenkette ein (alle Zeichen ";
PRINT "sind groß) ";
LINE INPUT ":",S$
' Schalte Caps Lock aus.
CapsAus
DruckeMeldung 24,1,"                    "
drutext$="Taste druecken um fortzufahren ..."
DruckeMeldung 25,1 drutext$
Do WHILE INKEY$="" : LOOP
CLS
END
' Schalte Caps Lock ein.
SUB CapsEin STATIC
   ' Setze Segment in untersten Speicherbereich.
   DEF SEG = 0
   ' Setze Caps Lock auf ein (setze Bit 6 von
   ' &H0417).
   POKE &H0417,PEEK(&H0417) OR &H40
   ' Segment wiederherstellen.
   DEF SEG
END SUB
```

```
' Schalte Caps Lock aus.
SUB CapsAus STATIC
   DEF SEG=0
   ' Setze Caps Lock auf aus (lösche Bit 6 von
   ' &H0417).
   POKE &H0417,PEEK(&H0417) OR &HBF
   DEF SEG
END SUB
' Ausgabe einer Meldung in Spalte%, Zeile%, ohne die
' Position des Cursors zu verändern.
SUB DruckeMeldung (Zeile%, Spalte%, Meldung$) STATIC
   ' Sichern der aktuellen Cursor-Position.
   CurZeile%=CSRLIN : CurSpalte%=Pos(0)
   LOCATE Zeile%, Spalte% : PRINT Meldung$;
   ' Bringe Cursor in alte Postion.
   LOCATE CurZeile%,CurSpalte%
END SUB
```

DEF*Typ*-Anweisungen

Funktion

Setzt den Standarddatentyp für Variablen, **DEF FN**-Funktionen und **FUNCTION**-Prozeduren.

Syntax

DEFINT *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFSNG *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFDBL *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFLNG *Buchstabenbereich* [, *Buchstabenbereich*]...
DEFSTR *Buchstabenbereich* [, *Buchstabenbereich*]...

Anmerkungen

Der *Buchstabenbereich* hat folgende Form:

Buchstabe1 [*-Buchstabe2*]

wobei *Buchstabe1* und *Buchstabe2* jeder Groß- oder Kleinbuchstabe des Alphabets sein können. Namen, die mit den in *Buchstabenbereich* angegebenen Buchstaben beginnen, gehören zu dem Typ, der durch die letzten drei Buchstaben der Anweisung bestimmt wird: ganzzahlig (**INT**), 4-Byte-Ganzzahl (**LNG**), einfache Genauigkeit (**SNG**), doppelte Genauigkeit (**DBL**) oder Zeichenkette (**STR**). Zum Beispiel ist im folgenden Ausschnitt `Meldung` eine Zeichenkettenvariable:

```
DEFSTR A-Q
.
.
.
Meldung="Platz auf dem Stapel nicht ausreichend."
```

Die Schreibweise des Buchstabens innerhalb des *Buchstabenbereiches* ist nicht bedeutsam. Alle folgenden Anweisungen sind gleich:

```
DEFINT I-N
DEFINT i-n
DEFINT i-N
```

Ein Typdeklarationszeichen (%, &, !, # oder $) hat immer Vorrang vor einer **DEF***Typ*-Anweisung.

DEF*Typ*-Anweisungen haben keine Auswirkung auf Verbund-Elemente.

Hinweis `I!`, `I#`, `I&`, `I$` und `I%` sind verschiedene Variablen und jede kann einen anderen Wert enthalten.

Unterschiede zu BASICA

BASICA behandelt **DEF***Typ*-Anweisungen unterschiedlich. BASICA prüft eine Anweisung vor jeder Ausführung. Wenn die Anweisung eine Variable ohne expliziten Typ (gekennzeichnet durch !, #, &, $ oder %) enthält, benutzt der Interpreter den aktuellen Standardtyp.

Im folgenden Beispiel bestimmt BASICA beim Interpretieren von Zeile `20`, daß der aktuelle Standardtyp für Variablen, die mit `I` beginnen, ganzzahlig ist. Zeile `30` ändert den Standardtyp in einfache Genauigkeit. Wenn BASICA zur Zeile `20` zurückkehrt, überprüft es die Zeile erneut und benutzt `IFLAG` als eine Variable mit einfacher Genauigkeit.

```
10 DEFINT I
20 PRINT IFLAG
30 DEFSNG I
40 GOTO 20
```

Im Gegensatz dazu prüft BASIC den Text nur einmal. Daher kann der Typ einer Variablen, sobald diese in einer Programmzeile vorgekommen ist, nicht mehr geändert werden.

Beispiel

Siehe Beispiel für die Funktion **ABS**.

DIM-Anweisung

Funktion

Deklariert eine Variable und ordnet Speicherplatz zu.

Syntax

DIM [**SHARED**] *Variable* [(*Indizes*)][**AS** *Typ*][, *Variable* [(*Indizes*)][**AS** *Typ*]]...

Anmerkungen

Die folgende Liste beschreibt die Teile der Anweisung **DIM**:

Teil	*Beschreibung*
SHARED	Das wahlfreie Attribut **SHARED** ermöglicht es allen Prozeduren in einem Modul, Datenfelder und einfache Variablen zu teilen. Dies unterscheidet sich von der Anweisung **SHARED**, die sich nur auf die Variablen innerhalb einer einzigen **SUB** oder **FUNCTION** auswirkt.
Variable	Ein BASIC-Variablenname.

Teil	*Beschreibung*
Indizes	Die Dimensionen eines Datenfeldes. Es können mehrere Dimensionen deklariert werden. Die Index-Syntax wird weiter unten beschrieben.
AS *Typ*	Deklariert *Variablen* als elementar oder benutzerdefiniert. Die elementaren Typen sind **INTEGER**, **LONG**, **SINGLE**, **DOUBLE** und **STRING** (variabel oder fest).

Indizes in der **DIM**-Anweisung haben folgende Form:

[*unterer* **TO**] *oberer* [,[*unterer* **TO**] *oberer*]...

Das **TO**-Schlüsselwort bietet eine Möglichkeit, sowohl die untere als auch die obere Grenze der Indizes eines Datenfeldes anzugeben. Die folgenden Anweisungen sind gleichbedeutend (vorausgesetzt, es wurde keine **OPTION BASE**-Anweisung verwendet):

```
DIM A(8,3)
DIM A(0 TO 8, 0 TO 3)
DIM A(8,0 TO 3)
```

Mit dem Schlüsselwort **TO** sind Sie nicht länger auf positive Indizes begrenzt. Sie können **TO** verwenden, um jeden Bereich von Indizes zwischen -32.768 bis 32.767 festzulegen:

```
DIM A(-4 TO 10)
DIM B(-99 TO -5, -3 TO 0)
```

Falls Sie in Ihrem Programm ein Datenfeld verwenden, ohne dieses in eine **DIM**-Anweisung einzuschließen, ist der Maximalwert des/der Datenfeldindex/-indizes **10**. Wenn Sie einen Index verwenden, der größer ist als der angegebene maximale Wert, erfolgt die Fehlermeldung `Index außerhalb des Bereichs`.

Die **DIM**-Anweisung initialisiert alle Elemente eines numerischen Datenfeldes zu Null und alle Elemente eines Zeichenketten-Datenfeldes zu Nullzeichenketten. Die Felder der Verbund-Variablen, einschließlich der Zeichenketten mit fester Länge, werden zu Null initialisiert.

Die maximal zulässige Anzahl der Dimensionen in einer **DIM**-Anweisung ist 60.

Falls Sie versuchen, mit einer **DIM**-Anweisung eine Datenfeldvariable zu dimensionieren, der bereits die Standarddimension 10 zugewiesen wurde, erscheint die Fehlermeldung `Datenfeld bereits dimensioniert`. Es empfiehlt sich daher beim Programmieren, die erforderlichen **DIM**-Anweisungen an den Anfang eines Programms außerhalb aller Verarbeitungsschleifen zu setzen.

Statische und dynamische Datenfelder

Die Art, in der Sie ein Datenfeld deklarieren, bestimmt ebenso, ob es **$STATIC** (bei der Übersetzung des Programms zugewiesen) oder **$DYNAMIC** (zugewiesen während der Programmausführung) ist.

- Ein zuerst in einer **COMMON**-Anweisung definiertes Datenfeld ist **$DYNAMIC**.
- Implizit dimensionierte Datenfelder sind **$STATIC**.
- Mit numerischen Konstanten oder **CONST**-Anweisungskonstanten dimensionierte Datenfelder sind **$STATIC**.
- Datenfelder, die mit Variablen als Indizes dimensioniert sind, sind **$DYNAMIC**.

Die folgende Liste zeigt die verschiedenen Kombinationen und deren Ergebnisse:

Anweisung	*Ergebnis*
`DIM A(0 TO 9)`	A wird ein **$STATIC**-Datenfeld zugewiesen, wenn **$DYNAMIC** nicht wirksam ist.
`DIM A(MAXDIM)`	Wenn `MAXDIM` in einer **CONST**-Anweisung definiert ist, ist A ein **$STATIC**-Datenfeld. Wenn `MAXDIM` eine Variable ist, dann ist das Datenfeld ein **$DYNAMIC**-Datenfeld und wird nur zugewiesen, wenn das Programm die **DIM**-Anweisung erreicht.

Weitere Informationen zu statischen und dynamischen Datenfeldern finden Sie in Anhang C, "Metabefehle", und Kapitel 2, "Datentypen".

Hinweis Falls die Datenfeldgröße 64K überschreitet, das Datenfeld nicht dynamisch ist und die **/ah**-Option nicht benutzt wurde, können Sie die Fehlermeldung `Index außerhalb des Bereichs` oder `Datenfeld zu groß` erhalten. Reduzieren Sie die Größe des Datenfeldes oder machen Sie das Datenfeld dynamisch und benutzen Sie die **/ah**-Befehlszeilenoption.

Typ-Deklarationen

Zusätzlich zur Festlegung der Dimensionen eines Datenfeldes kann die **DIM**-Anweisung ebenso dazu benutzt werden, den Typ einer Variablen zu deklarieren. In der folgenden Anweisung wird zum Beispiel die Variable als Ganzzahl deklariert, obwohl kein Typdeklarationzeichen bzw. keine **DEFINT**-Anweisung vorhanden ist.

```
DIM AnzahlBytes AS INTEGER
```

Die **DIM**-Anweisung bietet einen Mechanimus, um spezielle Variablen als Verbund zu deklarieren. Im folgenden Beispiel ist die Variable `TopKarte` als Verbundvariable deklariert.

```
TYPE Karte
    Farbe AS STRING*9
    Wert AS INTEGER
END TYPE
DIM TopKarte AS Karte
```

Sie können auch Datenfelder mit Verbunden als Elemente deklarieren:

```
TYPE Karte
    Farbe AS STRING*9
    Wert AS INTEGER
END TYPE
DIM Spielkarten (1 TO 52) AS Karte
```

Unterschiede zu BASICA

BASICA führt eine **DIM**-Anweisung aus, wenn es die Anweisung im Programm liest. Das Datenfeld wird nur zugewiesen, wenn die Anweisung ausgeführt wird. Daher sind alle Datenfelder in BASICA dynamisch.

Vergleichen Sie auch

ERASE; **OPTION BASE**; **REDIM**; **SHARED**; Anhang C, "Metabefehle", Kapitel 2, "Datentypen".

Beispiel

Das folgende Beispiel errechnet das Maximum und Minimum von einer Reihe von Werten:

```
' Finde Maximum und Minimum von bis zu 20 Werten.
'
' Dimensioniere ein Datenfeld zur Aufnahme der Werte.
CONST MAXDIM=20
DIM A(1 TO MAXDIM)
```

```
' Benutze DIM, um zwei ganzzahlige Variablen zu
' deklarieren.
' Alle anderen Variablen sind SINGLE.
DIM NumWerte AS INTEGER, I AS INTEGER

' Lies die Werte ein.
NumWerte=0
Text$ = "Gib pro Zeile einen Wert ein. Schreibe ENDE "
Text$ = Text$ + "zum beenden."
PRINT Text$
DO
   INPUT A$
   IF UCASE$(A$)="ENDE" OR NumWerte>=MAXDIM THEN _
                                            EXIT DO

   NumWerte=NumWerte+1
   A(NumWerte)=VAL(A$)
LOOP

' Finde größten und kleinsten Wert.
If NumWerte>0 THEN
   Max=A(1)
   Min=A(1)
   FOR I=1 TO NumWerte
      IF A(I)>Max THEN Max=A(I)
      IF A(I)<Min THEN Min=A(I)
   NEXT I

   PRINT "Der groesste Wert ist ";Max;
   PRINT " Der kleinste Wert ist ";Min
ELSE
   PRINT "Zu wenig Werte."
END IF
```

Ausgabe

```
Gib pro Zeile einen Wert ein. Schreibe ENDE zum beenden.
? 23.2
? 11.3
? 1.6
? ENDE
Der groesste Wert ist 23.2  Der kleinste Wert ist 1.6
```

DO...LOOP-Anweisung

Funktion

Wiederholt einen Anweisungsblock (solange, bis eine gegebene Bedingung wahr ist, bzw. bis eine gegebene Bedingung wahr wird).

Syntax 1

DO

[Anweisungsblock]

LOOP [{**WHILE** | **UNTIL**} *Boolescher Ausdruck*]

Syntax 2

DO [{**WHILE** | **UNTIL**} *Boolescher Ausdruck*]

[Anweisungsblock]

LOOP

Anmerkungen

Die folgende Liste beschreibt die Argumente der **DO...LOOP**-Anweisung:

Argument	***Beschreibung***
Anweisungsblock	Eine oder mehrere zu wiederholende BASIC-Anweisungen.
Boolescher Ausdruck	Jeder Ausdruck, der wahr (nicht Null) oder falsch (Null) auswertet.

Sie können eine **DO...LOOP**-Anweisung anstelle einer **WHILE...WEND**-Anweisung benutzen. Die **DO...LOOP**-Anweisung ist vielseitiger, da sie eine Bedingung am Beginn oder am Ende einer Schleife testen kann.

Beispiele

Die beiden ersten, weiter unten folgenden Beispiele zeigen Ihnen, welche Auswirkung die Anordnung der Bedingung darauf hat, wie häufig der Anweisungsblock ausgeführt wird. Das dritte Beispiel zeigt, wann am Ende der Schleife zu testen ist, und präsentiert ein Sortier-Unterprogramm, bei dem eine Prüfung am Ende angebracht ist.

Im folgenden Beispiel wird der Test am Beginn der Schleife durchgeführt. Weil `I` niemals kleiner 10 ist, wird der Körper der Schleife (der Anweisungsblock) niemals ausgeführt.

```
' DO...LOOP mit Test am Beginn der Schleife.
' Ausgabe zeigt, daß die Schleife nicht ausgeführt
' wurde.
   I = 10
   PRINT "Wert von I zu Beginn der Schleife ist ";I
   DO WHILE I < 10
      I = I + 1
   LOOP
   PRINT "Wert von I am Ende der Schleife ist ";I
```

Ausgabe

```
Wert von I zu Beginn der Schleife ist  10
Wert von I zum Ende der Schleife ist  10
```

Das folgende Beispiel testet `I` am Ende der Schleife, so daß der Anweisungsblock mindestens einmal ausgeführt wird.

```
' DO...LOOP mit Test am Ende der Schleife.
' Ausgabe zeigt, daß die Schleife einmal ausgeführt
' wurde.
   I = 10
     DO
        PRINT "Wert von I zu Beginn der Schleife";
        PRINT " ist ";I
       I = I + 1
   LOOP WHILE I < 10
   PRINT "Wert von I am Ende der Schleife ist ";I
```

Ausgabe

```
Wert von I zu Beginn der Schleife ist 10
Wert von I am Ende der Schleife ist 11
```

Das folgende Sortier-Unterprogramm testet am Ende der Schleife, weil das vollständige Datenfeld mindestens einmal untersucht werden muß, um festzustellen, ob es sortiert ist.

Allgemein gilt, daß Sie eine Prüfung am Schleifenende nur dann vornehmen sollten, wenn Sie sicher sind, daß der Schleifenkörper mindestens einmal ausgeführt werden soll.

```
' Bubble-Sort-Unterroutine
'       DatFeld ist das zu sortierende Datenfeld
'       N ist die Anzahl der Elemente in DatFeld
'
'       Hinweis: Sort geht davon aus, daß das erste
'                Element von DatFeld DatFeld(1) ist.
' Definiere einen besonderen Wert um anzuzeigen, daß
' kein Tausch vorgenommen wurde.
CONST KEINTAUSCH=-1
SUB sort(DatFeld(1),N) STATIC
Limit=N
DO
   Tausch=KEINTAUSCH
   FOR I=1 TO Limit-1     'Bearbeitet einmal das
                          'Datenfeld.
      IF DatFeld(I) > DatFeld(I+1) THEN
         SWAP DatFeld(I),DatFeld(I+1)  'Tauscht Daten-
                                       'feldelemente.
         Tausch=I               'Index des zuletzt
       END IF                   'getauschten Elementes.
   NEXT I
   Limit=Tausch      'Nächster Schritt nur bis dahin,
                     'wo letzter Tausch stattfand.
LOOP UNTIL Tausch=KEINTAUSCH  'Sortiert, bis keine
                              'Elemente mehr getauscht
                              'wurden.
END SUB
```

DRAW-Anweisung

Funktion

Zeichnet ein durch *Zeichenkettenausdruck* definiertes Objekt.

Syntax

DRAW *Zeichenkettenausdruck*

Anmerkungen

Die Anweisung **DRAW** faßt viele Möglichkeiten der anderen Grafikanweisungen in einer Grafik-Makrosprache zusammen, die unter "Befehle zur Cursor-Bewegung" und "Befehle für Winkel, Farbe und Teilungsfaktor" weiter unten beschrieben wird. Die Makrosprache definiert eine Gruppe von Eigenschaften, die zur Beschreibung eines Bildes benutzt werden kann. Diese Eigenschaften umfassen Bewegung (Auf, Ab, Links, Rechts), Farbe, Rotationswinkel und Teilungsfaktor.

Der *Zeichenkettenausdruck* besteht aus diesen Makrobefehlen, die durch ein oder mehrere Leerzeichen voneinander getrennt werden.

Befehle zur Cursor-Bewegung

Folgende Vorsatzbefehle können jedem der Bewegungsbefehle vorausgehen:

Vorsatz	*Beschreibung*
B	Bewegt den Cursor, zeichnet aber keine Punkte.
N	Bewegt den Cursor, kehrt danach aber zur Ausgangsposition zurück.

Die nachstehend genannten Befehle legen die Bewegung in Einheiten fest. Die Standard-Einheitsgröße ist ein Punkt. Diese Größe kann durch den Befehl **S**, der den Teilungsfaktor setzt, geändert werden (**S** wird im nächsten Abschnitt unter "Befehle für Winkel, Farbe und Teilungsfaktor" beschrieben). Wird kein Argument angegeben, so wird der Cursor um eine Einheit bewegt.

Jeder Bewegungsbefehl fängt mit der Bewegung an der aktuellen Grafikposition an. Diese ist in der Regel die Koordinate des letzten Bildpunktes, der mit einem anderen Grafik-Makrobefehl gezeichnet wurde. Bei der Ausführung eines Programms ist die aktuelle Position standardmäßig die Mitte des Bildschirms.

Bewegung	*Beschreibung*
U [*n*]	Bewegt den Cursor *n* Einheiten nach oben.
D [*n*]	Bewegt den Cursor *n* Einheiten nach unten.
L [*n*]	Bewegt den Cursor *n* Einheiten nach links.
R [*n*]	Bewegt den Cursor *n* Einheiten nach rechts.
E [*n*]	Bewegt den Cursor *n* Einheiten diagonal nach rechts oben.
F [*n*]	Bewegt den Cursor *n* Einheiten diagonal nach rechts unten.
G [*n*]	Bewegt den Cursor *n* Einheiten diagonal nach links unten.

Bewegung	*Beschreibung*
H [*n*]	Bewegt den Cursor *n* Einheiten diagonal nach links oben.
M *x,y*	Bewegt den Cursor absolut oder relativ. Ist *x* ein Plus- (+) oder Minus-Zeichen (-) vorangestellt, so erfolgt die Bewegung relativ zum aktuellen Punkt, d. h. *x* und *y* werden zu den Koordinaten der aktuellen Grafikposition addiert (oder von ihnen subtrahiert) und mit dieser Position durch eine Gerade verbunden. Geht *x* kein Vorzeichen voraus, so ist die Bewegung absolut, d. h. von der aktuellen Cursorposition zum Punkt mit den Koordinaten *x, y* wird eine Gerade gezogen.

Befehle für Winkel, Farbe und Teilungsfaktor

Die folgenden Befehle lassen Sie das Aussehen einer Zeichnung verändern, indem sie diese drehen, die Farbe verändern oder den Maßstab neu setzen:

Befehl — *Beschreibung*

A *n*

Setze den Rotationswinkel *n*. Der Wert für *n* kann zwischen 0 und 3 liegen, wobei 0 für 0° , 1 für 90°, 2 für 180° und 3 für 70° steht.

Um 90° oder 270° gedrehte Figuren werden so dargestellt, daß sie auf einem Bildschirm mit dem Standardaspekt von 4/3 in derselben Größe wie mit 0° oder 180° erscheinen.

TA *n*

Drehe einen Winkel um *n* Grad; *n* muß im Bereich von -360 bis 360 liegen. Ist *n* positiv, so erfolgt die Drehung entgegen dem Uhrzeigersinn; ist *n* negativ, so erfolgt die Drehung im Uhrzeigersinn. Das folgende Beispiel benutzt **TA**, um Speichen zu zeichnen:

```
SCREEN 1
FOR D=0 TO 360  STEP 10
   DRAW "TA="+VARPTR$(D)+"NU50"
NEXT D
```

C *n*

Setze die Farbe *n*. Informationen über gültige Farben, Farbnummern und Farbattribute finden Sie unter den Stichwörtern **COLOR, PALETTE** und **SCREEN**.

Befehl	***Beschreibung***
S *n*	Setze den Teilungsfaktor *n*, der im Bereich zwischen 1 und 255 liegen kann. Der Teilungsfaktor, multipliziert mit den Strecken, die mit den Befehlen **U, D, L, R** oder relatives **M** gegeben werden, ergibt die eigentliche zurückgelegte Strecke.
P *Füllfarbe, Randfarbe*	*Füllfarbe* ist die Farbe für das Innere einer Figur, während *Randfarbe* die Farbe für den Rand der Figur ist. Das Ausfüllen mit Mustern wird in **DRAW** nicht unterstützt.
V	Alle folgenden **DRAW**-Befehle benutzen logische und nicht physikalische Koordinaten.

Befehl für Teilzeichenketten	***Beschreibung***
X *Zeichenketten-ausdruck*	Führt die Teilzeichenkette aus. Dieser leistungsstarke Befehl ermöglicht Ihnen die Ausführung einer zweiten Teilfolge aus einer Zeichenkette. Sie können einen Zeichenkettenausdruck einen zweiten ausführen lassen, der wiederum einen dritten Zeichenkettenausdruck ausführt usw. Numerische Argumente können Konstanten, wie 123, oder Variablennamen sein. QuickBASIC benötigt das Format `"X" + VARPTR$` *(Zeichenkettenausdruck)* dieses Befehls. (Siehe auch nachfolgend "Unterschiede zu BASICA".)

Unterschiede zu BASICA

Die Anweisung **DRAW** erfordert Änderungen von BASICA-Programmen, wenn sie mit QuickBASIC verwendet werden. Speziell benötigt der Compiler das **VARPTR$**-Format für Variablen. Anweisungen wie

```
DRAW "XA$"
```

und

```
DRAW "TA = Winkel"
```

(wobei `A$` und `Winkel` Variablen sind) sollten im Compiler geändert werden in

```
DRAW "X" + VARPTR$(A$)
```

und

```
DRAW "TA =" + VARPTR$ (WINKEL)
```

Den Befehl

"**X***Zeichenkettenausdruck*"

unterstützt der Compiler nicht. Sie können eine Teilzeichenkette jedoch ausführen, indem Sie an X die Zeichenform der Adresse anhängen. Beispielsweise sind die folgenden zwei Anweisungen gleichwertig. Die erste Anweisung funktioniert beim Interpreter und Compiler, während die zweite nur beim Interpreter funktioniert.

```
DRAW "X" + VARPTR$(A$)
DRAW "XA$"
```

Beispiele

Das nachfolgende Programm zeichnet ein violettes Dreieck und füllt es kobaltblau aus.

```
SCREEN 1
DRAW "C2"                'Farbe = violett
DRAW "F60 L120 E60"      'Zeichnet ein Dreieck
DRAW "BD30"              'Bewegt sich in das Dreieck
DRAW "P1,2"              'Füllt das Dreieck aus
```

Das folgende Beispiel zeigt verschiedene Möglichkeiten der Verwendung des Makrobefehls **M**: bei absoluter Bewegung und relativer Bewegung; mit Zeichenketten-Variablenargumenten und mit numerischen Variablenargumenten.

```
SCREEN 2
DRAW "M 50,80"
DRAW "M 80,50
DRAW "M+40,-20"
DRAW "M-40,-20"
DRAW "M-40,+20"
DRAW "M+40,+20"
'Mit einer Zeichenkettenvariablen:
X$ = "400" : Y$ = "190"
```

```
DRAW "M" + X$ + "," + Y$
'Mit einer numerischen Variablen (Beachte die beiden
'"="-Zeichen).
A = 300 : B = 120
DRAW "M=" + VARPTR$(A)+ ",=" + VARPTR$(B)
```

Das folgende Beispiel zeichnet, unter Verwendung der Funktion **TIME$**, eine Uhr auf den Bildschirm.

```
' Deklariere Prozedur.
DECLARE SUB ZiffBlatt(Min$)
' Wähle 640 x 200 Punkte hochauflösenden
' Grafikbildschirm.
SCREEN 2
DO
   ' Lösche den Bildschirm.
   CLS
   ' Nimm die Zeichenkette, die die Minutenwerte
   ' enthält.
   Min$ = MID$(TIME$,4,2)
   ' Zeichne das Zifferblatt
   ZiffBlatt Min$
   ' Warte, bis die Minute wechselt oder eine Taste
   ' betätigt wird.
   DO
      ' Gib die Zeit am oberen Bildschirm aus.
      LOCATE 2,37
      PRINT TIME$
      ' Prüfung, ob Taste betätigt.
      Test$ = INKEY$
   LOOP WHILE Min$ = MID$(TIME$,4,2) AND Test$ = ""
' Beende das Programm, wenn eine Taste gedrückt wird.
LOOP WHILE Test$ = ""
END
' Zeichne das Zifferblatt:
SUB ZiffBlatt (Min$) STATIC
   LOCATE 23,30
Meldung$ = "Drücken Sie eine beliebige Taste zum "
Meldung$ = Meldung$ + "Beenden"
```

```
   PRINT Meldung$
   CIRCLE (320,100),175
   ' Wandele Zeichenketten in Zahlen um.
Std = VAL(TIME$)
Min = VAL(Min$)
' Wandele Zahlen in Winkel um.
Klein = 360 - (30 * Std + Min/2)
BIG = 360 - (6*Min)
' Zeichne die Zeiger.
DRAW "TA=" + VARPTR$(Klein) + "Nu40"
DRAW "TA=" + VARPTR$(Gross) + "Nu70"
END SUB
```

END-Anweisung

Funktion

Beendet ein BASIC-Programm, eine BASIC-Prozedur oder einen BASIC-Block.

Syntax

END [{**DEF** | **FUNCTION** | **IF** | **SELECT** | **SUB** | **TYPE**}]

Anmerkungen

END DEF beendet eine mehrzeilige **DEF FN**-Funktionsdefinition. Sie müssen **END DEF** bei einer mehrzeiligen **DEF FN** verwenden.

Die **END FUNCTION**-Anweisung beendet eine **FUNCTION**-Prozedurdefinition. Sie müssen **END FUNCTION** bei **FUNCTION** verwenden.

Die **END IF**-Anweisung beendet eine Block-Anweisung **IF...THEN...ELSE**. Sie müssen **END IF** bei dem Block-**IF...THEN...ELSE** verwenden.

Die **END SELECT**-Anweisung beendet einen **SELECT CASE**-Block. Sie müssen **END SELECT** bei einer **SELECT CASE**-Anweisung benutzen.

Die **END SUB**-Anweisung beendet ein BASIC-Unterprogramm. Sie müssen **END SUB** bei **SUB** verwenden.

Die **END TYPE**-Anweisung beendet die Definition eines benutzerdefinierten Typs. Sie müssen **TYPE** mit **END TYPE** abschließen.

Die **END**-Anweisung selbst stoppt die Programmausführung und schließt alle Dateien. In allein ausführbaren Programmen gibt **END** die Kontrolle an das Betriebssystem zurück. Wenn ein Programm innerhalb der QuickBASIC-Umgebung läuft, führt **END** Sie zur Umgebung zurück.

Der Compiler nimmt am Ende jedes Programms eine **END**-Anweisung an, so daß das Weglassen der **END**-Anweisung am Programm-Ende immer noch eine korrekte Programmbeendigung zur Folge hat.

Sie können **END**-Anweisungen überall im Programm einsetzen, um seine Ausführung zu beenden.

Vergleichen Sie auch

DEF FN; FUNCTION; IF...THEN...ELSE; SELECT CASE; SUB; TYPE

Beispiel

Das folgende Beispiel benutzt ein Unterprogramm, um den Benutzer zu fragen, ob weitere Aktionen ausgeführt werden sollen. Wenn der Benutzer `n` oder `N` eingibt, beendet das Unterprogramm das Programm.

```
DO
   .
   .
   .
   CALL WeiterFrage
LOOP
SUB WeiterFrage STATIC
   DO
      INPUT "Weiter (J oder N)"; Antwort$
      R$=UCASE$(LEFT$(Antwort$,1))
      IF R$="N" THEN END
      IF R$="J" THEN EXIT DO
      BEEP
   LOOP
END SUB
```

ENVIRON-Anweisung

Funktion

Ändert einen Parameter in der DOS-Umgebungszeichenketten-Tabelle.

Syntax

ENVRIRON *Zeichenkettenausdruck*

Anmerkungen

Der *Zeichenkettenausdruck* muß die Form *Parameterid=Text* oder die Form *Parameterid Text* haben. Alles, was links vom Gleichheitszeichen oder vom Leerzeichen steht, wird als Parameter angenommen, und alles, was rechts steht, als Text.

Falls es vorher in der Umgebungszeichenketten-Tabelle keine *Parameterid* gab, wird sie am Ende der Tabelle hinzugefügt. Gibt es bei der Ausführung der Anweisung **ENVIRON** in der Tabelle eine *Parameterid,* so wird diese gelöscht und die neue *Parameterid* am Ende der Tabelle hinzugefügt.

Die Textzeichenkette ist der neue Parametertext. Wenn dieser Text eine Nullzeichenkette ("") oder ein Semikolon (";") ist, wird der vorhandene Parameter aus der Umgebungszeichenketten-Tabelle entfernt und der verbleibende Hauptteil der Datei verdichtet.

Wenn Sie Ihr Programm beenden, entfernt DOS die von dieser Funktion geänderte Umgebungszeichenketten-Tabelle. Die Umgebungszeichenketten-Tabelle ist die vor dem Programmlauf gültige.

Mit dieser Anweisung können Sie den Parameter **PATH** für eine Zwischenbearbeitung ("Child Process"), ein Programm oder Befehl, der mit einer **SHELL**-Anweisung gestartet wird) ändern oder Parameter durch die Einführung eines neuen Umgebungsparameters an eine Zwischenstufe übergeben.

Zu den Fehlern in Umgebungszeichenketten-Tabellen gehören Parameter, die keine Zeichenketten sind und zu wenig freien Platz haben. Wenn der Umgebungszeichenketten-Tabelle kein Platz mehr zugeordnet werden kann, erfolgt die Fehlermeldung `Speicherkapazität reicht nicht aus`. Der freie Platz in der Tabelle ist in der Regel ziemlich knapp bemessen.

Vergleichen Sie auch

ENVIRON$, SHELL

Beispiel

```
'Ändert den Wert der PFAD-Umgebungsvariablen.
ENVIRON "PATH=A:\VERKAUF;A:\RECHNWES"
```

ENVIRON$-Funktion

Funktion

Stellt eine Umgebungszeichenkette aus der DOS-Umgebungszeichenketten-Tabelle bereit.

Syntax

ENVIRON$ (*Umgebungszeichenkette*)

ENVIRON$ (*n*)

Anmerkungen

Die *Umgebungszeichenkette* ist eine Zeichenketten-Konstante oder -Variable, die den Namen der Umgebungsvariablen enthält. Das Argument *n* ist ein numerischer Ausdruck.

Wenn Sie einen *Umgebungszeichenketten*-Namen festlegen, der jedoch entweder in der BASIC-Umgebungszeichenketten-Tabelle nicht gefunden werden kann oder dem kein Text nachfolgt, gibt **ENVIRON$** eine Nullzeichenkette an. Andernfalls gibt **ENVIRON$** den in der Umgebungszeichenketten-Tabelle auf das Gleichheitszeichen folgenden Text zurück.

Bei Angabe eines numerischen Arguments (*n*) wird die *n*-te Zeichenkette in der Umgebungszeichenketten-Tabelle ermittelt. In einem solchen Fall umfaßt die Zeichenkette den gesamten Text, einschließlich des *Umgebungszeichenketten*-Namens. Falls keine *n*-te Zeichenkette existiert, gibt **ENVIRON$** eine Nullzeichenkette an. Das Argument *n* kann jeder beliebige numerische Ausdruck sein; er wird auf eine ganze Zahl gerundet.

Beispiel

Das folgende Beispiel gibt die aktuellen Einstellungen der Umgebungszeichenketten-Tabelle aus:

```
I = 1
DO WHILE ENVIRON$(I) <> ""
   PRINT ENVIRON$(I)
   I = I + 1
LOOP
```

Ausgabe

```
COMSPEC=C:\COMMAND.COM
TMP=c:\tmp
PATH=C:\TOOLS;C:\BIN
PROMPT=$ec4$l$ec7$p$ec4$g$ec7$eb1
INIT=c:\tools
LIB=c:\lib
INLUDE=c:\include
```

EOF-Funktion

Funktion

Prüft auf die Dateiende-Bedingung.

Syntax

EOF (*Dateinummer*)

Anmerkungen

Wenn das Ende einer sequentiellen Datei erreicht wurde, gibt die **EOF**-Funktion **EOF** -1 (wahr) an. Benutzen Sie die **EOF**-Funktion, um auf Dateiende zu prüfen, während Daten eingelesen werden. Auf diese Weise vermeiden Sie die Fehlermeldung `Eingabe nach logischem Ende`.

Wenn **EOF** bei Direktzugriffs- oder Binär-Dateien benutzt wird, gibt sie "wahr" zurück, wenn die zuletzt ausgeführte **GET**-Anweisung keinen ganzen Datensatz einlesen konnte. Der Grund hierfür ist ein Versuch, über das Dateiende-Zeichen hinaus zu lesen.

EOF kann nicht mit den BASIC-Geräten **SCRN:**, **KYBD:**, **CONS:** und **LPT***n***:** benutzt werden.

Wenn Sie **EOF** bei einem Datenübertragungsgerät einsetzen, hängt die Definition der Dateiende-Bedingung von dem Modus (ASCII oder Binär) ab, in dem Sie das Gerät aktiviert haben. Im ASCII-Modus ist **EOF** solange falsch, bis Sie STRG+Z eingeben; danach bleibt es wahr, bis Sie das Gerät deaktivieren. Im Binärmodus ist **EOF** wahr, wenn die Eingabe-Warteschlange leer ist (**LOC(***n***)=0**). **EOF** wird falsch, wenn die Eingabe-Warteschlange nicht leer ist.

Beispiel

Der folgende Ausschnitt liest Werte mit einfacher Genauigkeit aus einer Datei, bis alle Werte gelesen sind.

```
DIM M (0 TO 2000)
OPEN "DATEN" FOR INPUT AS 1
C = 0
DO WHILE NOT EOF(1) AND C <= 2000
      INPUT #1, M(C)
      C = C + 1
LOOP
.
.
.
```

ERASE-Anweisung

Funktion

Initialisiert die Elemente von statischen Datenfeldern oder hebt die Zuordnung für dynamische Felder auf.

Syntax

ERASE *Datenfeldname* [, *Datenfeldname*...]

Anmerkungen

Die *Datenfeldname*-Argumente sind die Namen der zu löschenden Datenfelder.

ERASE hat jeweils eine andere Wirkung auf **$STATIC**- und **$DYNAMIC**-Datenfelder.

Die Anweisung **ERASE** setzt die Elemente eines **$STATIC**-Datenfeldes auf Null, wenn es ein numerisches Datenfeld, oder auf Nullzeichenkette (""), wenn es ein Zeichenketten-Datenfeld ist. Wenn das Datenfeld ein Verbund-Datenfeld ist, setzt die Anweisung **ERASE** alle Elemente jedes einzelnen Datenfeldes auf Null, einschließlich der Zeichenketten-Elemente mit fester Länge.

Die Benutzung von **ERASE** für ein **$DYNAMIC**-Datenfeld jedoch gibt den vom Datenfeld benutzten Speicher wieder frei. Bevor Ihr Programm sich wieder auf das **$DYNAMIC**-Datenfeld beziehen kann, müssen Sie zuerst das Datenfeld mit einer **DIM**- oder **REDIM**-Anweisung neu dimensionieren. Neudimensionierung eines Datenfeldes ohne vorheriges Löschen führt zu einem Laufzeit-Fehler `Neu dimensioniertes Datenfeld.` Die **ERASE**-Anweisung ist nicht erforderlich, wenn Datenfelder mit **REDIM** neu dimensioniert werden.

Weitere Informationen zur Deklaration von **$DYNAMIC**- und **$STATIC**-Datenfeldern finden Sie in Kapitel 2, "Datentypen", und im Anhang C, "Metabefehle".

Vergleichen Sie auch

DIM, REDIM

Beispiel

Der folgende Programmausschnitt zeigt den Gebrauch von **ERASE** mit den **$DYNAMIC**- und **$STATIC**-Metabefehlen.

```
REM $DYNAMIC
DIM A(100,100)
.
.
.
'Speicherzuordnung für Datenfeld A wird aufgehoben
ERASE A
'Dimensioniert Datenfeld A erneut
```

```
REDIM A(5,5)
REM $STATIC
DIM B(50,50)
.
.
.
'Dies setzt alle Elemente von B gleich Null; B hat
'immer noch die Dimensionierung von DIM.
ERASE B
```

ERDEV-, ERDEV$-Funktionen

Funktion

Bietet gerätespezifische Statusinformationen nach einem Fehler.

Syntax

ERDEV

ERDEV$

Anmerkungen

ERDEV ist eine Ganzzahl-Funktion, die einen Fehlercode von dem letzten Gerät, welches einen Fehler verursacht hat, angibt. **ERDEV$** ist eine Zeichenkettenfunktion, die den Namen des Gerätes angibt, das das Auftreten eines Fehlers verursacht hat. Weil **ERDEV** und **ERDEV$** sinnvolle Informationen nur nach dem Auftreten eines Fehlers ausgeben, werden sie gewöhnlich bei der Fehlerbehandlung in einer **ON ERROR**-Anweisung benutzt.

ERDEV und **ERDEV$** dürfen nicht auf der linken Seite einer Zuweisung stehen.

ERDEV wird von der Behandlungsroutine Kritischer Fehler (Critical Error Handler; Interrupt 24H) gesetzt, wenn DOS einen Fehler entdeckt, der die weitere Ausführung verhindert.

Der Wert von **ERDEV** wird binär angegeben und enthält die DOS-Fehlerinformation. Die niederwertigen 8 Bits (erstes Byte) enthalten den DOS-Fehlercode, ein Wert von 0 bis 12. Die höherwertigen 8 Bits (zweites Byte) enthalten die Bits 15, 14, 13, XX, 3, 2, 1 und 0 in der Reihenfolge des Geräteattribut-Wortes. Das mit XX angezeigte Bit ist immer Null. Weitere Informationen zu Geräteattribut-Worten finden Sie im *Microsoft MS-DOS® Programmer's Reference.*

Vergleichen Sie auch

ERL, ERR, ON ERROR

Beispiele

Das folgende Beispiel gibt die Werte von **ERDEV** und **ERDEV$** nach einem Fehler aus.

```
DEFINT A-Z
' Markiere erste Zeile der Fehlerbehandlung.
ON ERROR GOTO FehlerBehandlung
' Versuch, die Datei zu öffnen.
OPEN "A:JUNK.DAT" FOR INPUT AS #1
END
' Fehlerbehandlungsroutine. Schreibt die Werte von ' ERDEV
und ERDEV$.
FehlerBehandlung:
   PRINT "ERDEV Wert ist ";ERDEV
   PRINT "Geräte-Name ist ";ERDEV$
   ON ERROR GOTO 0
```

Ein Programmlauf mit geöffnetem Laufwerk A verursacht die folgende Ausgabe (2 ist der Fehlercode für `Laufwerk nicht bereit`):

```
ERDEV Wert ist    2
Geräte-Name ist A:
```

ERR-, ERL-Funktionen

Funktion

Geben den Fehlerstatus an.

Syntax

ERR

ERL

Anmerkungen

Nach einem Fehler gibt die Funktion **ERR** den Fehlercode und die Funktion **ERL** die Zeilennummer, in der der Fehler aufgetreten ist, an. Da **ERR** und **ERL** sinnvolle Informationen nur nach dem Auftreten eines Fehlers ausgeben, werden sie gewöhnlich in Fehlerbehandlungsroutinen verwendet, um den Fehler und dessen korrekte Behandlung zu bestimmen.

Da **ERL** und **ERR** Funktionen sind, können sie nicht auf der linken Seite einer Zuweisung stehen. Trotzdem können Sie sie indirekt mit der **ERROR**-Anweisung setzen.

Unterschiede zu BASICA

Die **ERL**-Funktion gibt nur die Zeilennummer und nicht die Zeilenmarke aus, die vor der fehlerverursachenden Zeile liegt. Wenn Ihr Programm keine Zeilennummern hat, gibt **ERL** immer 0 an.

Vergleichen Sie auch

ERDEV, ERROR, ON ERROR, RESUME

Beispiel

Siehe Beispiel für **ON ERROR.**

ERROR-Anweisung

Funktion

Simuliert das Auftreten eines BASIC-Fehlers oder ermöglicht dem Benutzer die Definition von Fehlercodes.

Syntax

ERROR *Ganzzahlausdruck*

Anmerkungen

Der *Ganzzahlausdruck* repräsentiert einen Fehlercode. Er muß größer als 0 und kleiner gleich als 256 sein. Wenn der *Ganzzahlausdruck* gleich einem Fehlercode ist, der bereits von BASIC benutzt wird, simuliert die Anweisung **ERROR** das Vorkommen dieses Fehlers und gibt die entsprechende Fehlermeldung aus. (Eine vollständige Liste der von BASIC benutzten Fehlercodes finden Sie in Anhang D, "Fehlermeldungen".)

Zur Definition Ihres eigenen Fehlercodes verwenden Sie einen Wert, der größer ist als irgendeiner der BASIC-Standardfehlercodes. (Beginnen Sie mit 255 und zählen Sie zurück, um eine Kompatibilität mit zukünftigen Microsoft BASIC-Fehlercodes zu erreichen.)

Gibt eine **ERROR**-Anweisung einen Code an, für den keine Fehlermeldung definiert ist, so wird die Meldung

```
Nichtdruckbarer Fehler
```

ausgegeben. Die Ausführung einer **ERROR**-Anweisung, für die es keine Fehlerbehandlungsroutine gibt, bewirkt die Ausgabe einer Fehlermeldung und das Anhalten des Programms.

Vergleichen Sie auch

ERL, ERR, ON ERROR, RESUME

Beispiel

Das folgende Programmfragment benutzt eine **ERROR**-Anweisung um einen Benutzereingabefehler abzufangen:

```
ON ERROR GOTO Behandlung
OeffneDatei:
   PRINT "Name der zu aktualisierenden Datei";
   INPUT ":";DateiName$
   IF DateiName$ = "" THEN END
   OPEN DateiName$ FOR INPUT AS #1
   PRINT "Die ersten vier Zeilen von ";DateiName$;
   PRINT "sind:"
   PRINT
   FOR I = 1 TO 4
        LINE INPUT #1, Temp$
        PRINT Temp$
   NEXT
```

```
   PRINT : INPUT "Ist dieses die richtige Datei";R$
   'Definiert Fehler 200
   IF LEFT$(R$,1) <> "J" THEN ERROR 200
   .
   .
   .
   END
Behandlung:          'Fehlerbehandlungsroutine
   Nummer = ERR
   'Laufzeit-Fehler für "Datei nicht gefunden"
   IF Nummer = 53 THEN
      CLOSE #1
      PRINT "Datei nicht in diesem Verzeichnis"
      Meld$ = "Einen neuen Dateinamen eingeben"
      Meld$ = Meld$ + "([d:]\xxx\...\xxx) oder"
      PRINT Meld$
      Meld$ = "<EINGABETASTE> betätigen, um das Programm zu"
      PRINT Meld$ + " beenden"
      RESUME OeffneDatei
   ELSEIF Nummer = 200 THEN   'Benutzereingabe "n"
      CLOSE #1
      RESUME OeffneDatei
   ELSE
      ERROR Zahl          'Anderer Fehler als 53 oder 200
      ON ERROR GOTO 0     'Meldung ausgeben,Fehler-
   END IF                 'behandlung aufheben und
                          'Programm stoppen
```

Ausgabe

```
Name der zu aktualisierenden Datei? c:\nov\manf.txt
Datei nicht in diesem Verzeichnis
Einen neuen Dateinamen eingeben ([d:]\xxx\...\xxx) oder
<EINGABETASTE> betätigen, um das Programm zu beenden
Name der zu aktualisierenden Datei? c:\hum\majork.txt
Die ersten vier Zeilen von c:
Ich liebe die Berge,
die Gletscher, den Schnee;
beim Skifahren, da tun mir
die Füße oft weh.
Ist dies die richtige Datei? J
```

EXIT-Anweisung

Funktion

Beendet eine **DEF FN**-Funktion, **DO...LOOP**-, **FOR...NEXT**-Schleife, **FUNCTION** oder **SUB**.

Syntax

EXIT {**DEF** | **DO** | **FOR** | **FUNCTION** | **SUB**}

Anmerkungen

Anweisung	*Beschreibung*
EXIT DEF	Verläßt eine **DEF FN**-Funktion.
EXIT DO	Ermöglicht ein alternatives Verlassen einer **DO...LOOP**-Schleife.
EXIT FOR	Bietet einen weiteren Weg, eine **FOR...NEXT**-Schleife zu verlassen.
EXIT FUNCTION	Verläßt eine **FUNCTION**-Prozedur.
EXIT SUB	Verläßt ein Unterprogramm.

Keine der **EXIT**-Anweisungen definiert das Ende der Struktur, in der sie benutzt wird. **EXIT**-Anweisungen bieten nur ein alternatives Verlassen der Struktur.

Die folgende Liste beschreibt die Anweisungen im Detail:

- **EXIT DEF**

  EXIT DEF bewirkt ein sofortiges Verlassen der ausführenden **DEF FN**-Funktion. Die Programmausführung wird an der Stelle fortgesetzt, an der die **DEF FN**-Funktion aufgerufen wurde.

- **EXIT DO**

  Die Anweisung **EXIT DO** kann nur innerhalb einer **DO...LOOP**-Anweisung verwendet werden. **EXIT DO** übergibt die Kontrolle an die Anweisung, die der **LOOP**-Anweisung folgt.

  Bei verschachtelten **DO...LOOP**-Anweisungen führt eine **EXIT DO**-Anweisung aus der sie unmittelbar umschließenden Schleife heraus.

- **EXIT FOR**

  Eine **EXIT FOR**-Anweisung darf nur in einer **FOR...NEXT**-Schleife erscheinen. **EXIT FOR** übergibt die Kontrolle an die Anweisung, die der **NEXT**-Anweisung folgt.

  Bei verschachtelten **FOR...NEXT**-Schleifen führt eine **EXIT FOR**-Anweisung aus der sie unmittelbar umschließenden Schleife heraus.

- **EXIT FUNCTION**

  Die **EXIT FUNCTION**-Anweisung bewirkt ein sofortiges Verlassen einer **FUNCTION**-Prozedur. Die Programmausführung wird dort fortgesetzt, wo **FUNCTION** aufgerufen wurde.

  EXIT FUNCTION kann nur in einer **FUNCTION**-Prozedur verwendet werden.

- **EXIT SUB**

  Die Anweisung **EXIT SUB** verläßt eine **SUB**-Prozedur sofort. Die Programmausführung fährt mit der Anweisung nach der **CALL**-Anweisung fort.

  EXIT SUB kann nur in einer **SUB**-Prozedur benutzt werden.

Vergleichen Sie auch

DEF FN; DO...LOOP; FOR...NEXT; FUNCTION; SUB

Beispiele

Für die Benutzung von **EXIT SUB** siehe drittes Beispiel von **STATIC**.

Das folgende Unterprogramm ist ein erweitertes **RTRIM$,** das folgende Leerzeichen, Tabulatoren, Wagenrückläufe und Zeilenvorschübe aus einer Zeichenkette entfernt. Das Unterprogramm beginnt die Suche am Ende der Zeichenkette und benutzt **EXIT FOR**, um aus der Schleife herauszuspringen, wenn das erste Druckzeichen gefunden ist.

```
' Rtrim entfernt nachfolgende Blanks, Tabs,
' Wagenrückläufe und Zeilenvorschübe aus einer
' Zeichenkette.
SUB Rtrim(S$) STATIC
J=0
' Beginne am Ende der Zeichenkette und finde das
' erste Zeichen, das kein Blank, Tab, Wagenrücklauf
' oder Zeilenvorschub ist.
```

```
FOR I=LEN(S$) TO 1 STEP -1
   C$=MID$(S$,I,1)
   IF C$<>" " AND C$<>CHR$(9) AND C$<>CHR$(10) _
                              AND C$<>CHR$(13) THEN
      J=I
      EXIT FOR
   END IF
NEXT I
' Entferne die unerwünschten nachfolgenden Zeichen.
S$=LEFT$(S$,J)
END SUB
```

EXP-Funktion

Funktion

Berechnet die Exponentialfunktion.

Syntax

EXP (*x*)

Anmerkungen

Die **EXP**-Funktion gibt den Wert von *e* (die Basis des natürlichen Logarithmus) potenziert mit *x* an. Der Exponent *x* muß kleiner oder gleich 88,02969 sein. Wenn *x* größer als 88,02969 ist, erscheint die Fehlermeldung `Überlauf`.

EXP wird standardmäßig mit einfacher Genauigkeit berechnet; wenn das Argument *x* von doppelter Genauigkeit ist, wird **EXP** mit doppelter Genauigkeit berechnet.

Vergleichen Sie auch

LOG

Beispiel

Das nachstehende Programm benutzt die Funktion **EXP**, um das Wachstum einer Bakterienkolonie über einen Zeitraum von 15 Tagen zu berechnen. Da das Wachstum der Population von der sich ständig ändernden Größe abhängt, verläuft es exponential.

```
INPUT "Anfängliche Bakterienpopulation"; Kolonie0
PRINT "Wachstumsrate pro Tag in Prozent der ";
INPUT "Population"; Rate
R = Rate/100 : Form$="##           ###,###"
PRINT : PRINT "Tag             Population" : PRINT
FOR T = 0 TO 15 STEP 5
    PRINT USING Form$; T, Kolonie0 * EXP (R*T)
NEXT
```

Ausgabe

```
Anfängliche Bakterienpopulation? 10000
Wachstumsrate pro Tag in Prozent der Population? 10
Tag          Population
 0           10,000
 5           16,487
10           27,183
15           44,817
```

FIELD-Anweisung

Funktion

Weist Platz für Variablen in einem Direktzugriffs-Dateipuffer zu.

Syntax

FIELD [#] *Dateinummer*, *Feldlänge* **AS** *Zeichenkettenvariable...*

Anmerkungen

Hinweis Die Verbundvariablen und die Syntax der erweiterte **OPEN**-Anweisung bieten einen bequemeren Weg, Direktzugriffsdateien zu benutzen. In Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie eine eingehendere Diskussion der Verwendung von Verbundvariablen für Datei-E/A.

Die folgende Liste beschreibt die Argumente der **FIELD**-Anweisungen.

Argument	***Beschreibung***
Dateinummer	Die Nummer, unter der die Datei in einer **OPEN**-Anweisung eröffnet wurde.
Feldlänge	Die Länge des Feldes in dem Satz.
Zeichenketten-variable	Die Zeichenkettenvariable, die die von einem Satz oder einer Datei gelesenen Daten enthält und die in einer Zuweisung benutzt wird, wenn Informationen in den Satz geschrieben werden.

Die Gesamtzahl der Bytes, die Sie in einer **FIELD**-Anweisung zuordnen, darf die beim Eröffnen der Datei festgelegte Datensatzlänge nicht überschreiten. Andernfalls erfolgt die Fehlermeldung `FIELD-Überlauf`. (Die Standardsatzlänge beträgt 128 Bytes.)

Für dieselbe Datei kann eine beliebige Anzahl von **FIELD**-Anweisungen ausgeführt werden. Alle ausgeführten **FIELD**-Anweisungen bleiben gleichzeitig wirksam.

Beim Schließen der Datei werden alle Felddefinitionen für eine Datei gelöscht, d. h. alle als Felder mit der Datei definierten Zeichenketten werden auf Null gesetzt.

Wichtig Benutzen Sie in einer **INPUT**- oder Zuweisungsanweisung keinen als Feld definierten Variablennamen, wenn die Variable mit **FIELD** zugeordnet bleiben soll. Sobald ein Variablenname als Feld zugeordnet worden ist, zeigt er auf die richtige Stelle im Direktzugriffs-Dateipuffer. Wenn eine nachfolgende **INPUT**- oder Zuweisungsanweisung mit diesem Variablennamen ausgeführt wird, bezieht sich der Zeiger der Variablen nicht mehr auf den Direktzugriffs-Datensatzpuffer, sondern auf den Zeichenkettenraum.

Unterschiede zu BASICA

Wenn eine Direktzugriffsdatei in einem kompilierten Programm mit einer **CLOSE**- oder **RESET**-Anweisung geschlossen wird, werden alle mit dieser Datei verbundenen und mit **FIELD** zugeordneten Variablen auf Null-Zeichenketten zurückgesetzt. Wird eine Direktzugriffsdatei in einem BASICA-Programm geschlossen, so behalten die mit **FIELD** zugeordneten Variablen den letzten Wert, der ihnen durch eine **GET**-Anweisung zugewiesen wurde.

Vergleichen Sie auch

GET, LSET, OPEN, PUT, RSET

Beispiel

Dieses Beispiel zeigt einen mehrfach definierten Direktzugriffs-Dateipuffer. In der ersten **FIELD**-Anweisung wird der 66-Byte-Puffer in 6 getrennte Variablen für Name, Adresse, Stadt, Staat und Postleitzahl aufgeteilt. In der zweiten **FIELD**-Anweisung wird derselbe Puffer im ganzen einer einzigen Variablen `PLIST$` zugewiesen. Zum Schluß prüft das Programm, ob die Variable `PLZ$`, welche die Postleitzahl enthält, in einen bestimmten Bereich fällt; wenn ja, gibt es die vollständige Adreßzeichenkette aus.

```
' Definiert Feld- und Verbundlängen mit Konstanten.
CONST VNAML=10, LNAML=15, ADRL=25, STDL=10, STL=2
CONST PLZL=4
CONST RECLEN=VNAML+LNAML+ADRL+STDL+STL+PLZL
OPEN "ADRESSLISTE" FOR RANDOM AS #1 LEN=RECLEN
FIELD #1, VNAML AS Vnam$, LNAML AS Lnam$, _
   ADRL AS Adr$, STDL AS Std$, STL AS St$, _
   PLZL AS Plz$
FIELD #1, RECLEN AS Plist$
GET #1, 1
' Lies die Datei, suche nach Postleitzahlen im Bereich
' 4000 bis 4800.
DO WHILE NOT EOF(1)
   Zcheck$ = Plz$
   IF (Zcheck$ > "3999" AND ZCHECK$ < "4801") THEN
      Info$ = Plist$
      PRINT LEFT$(Info$,25)
      PRINT MID$(Info$,26,25)
      PRINT RIGHT$(Info$,17)
   END IF
   GET #1
LOOP
```

FILEATTR-Funktion

Funktion

Gibt Informationen über eine geöffnete Datei.

Syntax

FILEATTR (*Dateinummer, Attribut*)

Anmerkungen

Die **FILEATTR**-Funktion hat folgende Argumente:

Argument	*Beschreibung*
Dateinummer	Die Nummer einer geöffneten Datei. Es ist die gleiche Nummer, die in der Anweisung **OPEN** benutzt wurde. Sie können einen numerischen Ausdruck solange benutzen, wie er die Nummer einer geöffneten Datei berechnet.
Attribut	Zeigt den Typ der zurückzugebenden Information an. Wenn *Attribut* den Wert 1 hat, gibt **FILEATTR** eine Kodierung an, die den Dateimodus anzeigt (siehe unten). Wenn *Attribut* den Wert 2 hat, gibt **FILEATTR** die DOS-Dateibehandlung der Datei an.

Die folgende Tabelle listet die Rückgabewerte und zugehörigen Dateimodi, wenn *Attribut* den Wert 1 hat.

Rückgabewert	*Modus*
1	**INPUT**
2	**OUTPUT**
4	**RANDOM**
8	**APPEND**
32	**BINARY**

Vergleichen Sie auch

OPEN

Beispiel

Das folgende Beispiel öffnet zwei Dateien und druckt die DOS-Dateibehandlung und -Dateimodi, die von **FILEATTR** angegeben werden, aus:

```
OPEN "tempfile.dat" FOR APPEND AS #1
OPEN "tempfl2.dat" FOR RANDOM AS #2
PRINT "Zahl  Dateibehandlung  Dateimodus"
PRINT TAB(2);1;TAB(13);FILEATTR(1,2);
PRINT TAB(28);FILEATTR(1,1)
PRINT TAB(2);2;TAB(13);FILEATTR(2,2);
PRINT TAB(28);FILEATTR(2,1)
END
```

Ausgabe

```
Zahl  Dateibehandlung  Dateimodus
1         5               8
2         6               4
```

FILES-Anweisung

Funktion

Druckt die Namen der Dateien aus, die sich auf der angegebenen Diskette bzw. Festplatte befinden.

Syntax

FILES [*Dateiangabe*]

Anmerkungen

Die *Dateiangabe* ist eine Zeichenkettenvariable oder -konstante, die entweder einen Dateinamen oder einen Verzeichnisnamen und wahlweise eine Laufwerksangabe enthält.

Wenn Sie *Dateiangabe* auslassen, listet die **FILES**-Anweisung sämtliche Dateien des aktuellen Verzeichnisses auf. Sie können die DOS-Globalzeichen - Fragezeichen (?) oder Stern (*) - benutzen. Ein Fragezeichen entspricht jedem einzelnen Zeichen im Dateinamen oder in der Erweiterung. Ein Stern entspricht einem oder mehreren Zeichen, beginnend an dieser Position.

Falls Sie eine *Dateiangabe* ohne einen expliziten Pfad benutzen, ist das aktuelle Verzeichnis das Standardverzeichnis.

Bitte beachten Sie, daß der von **FILES** ausgegebene Kennsatz, ungeachtet des in *Dateiangabe* enthaltenen Verzeichnisnamens, immer das aktuelle Verzeichnis ist.

Beispiele

```
FILES                 'Zeigt alle Dateien im aktuellen
                      'Verzeichnis
FILES "*.BAS"         'Zeigt alle Dateien mit der
                      'Erweiterung .BAS
FILES "B:*.*"         'Zeigt alle Dateien in Laufwerk B
FILES "B:"            'Entspricht "B:*.*"
FILES "Test?.BAS"     'Zeigt alle Dateien mit fünf
                      'Buchstaben, die mit "TEST"
                      'beginnen und die Erweiterung .BAS
                      'haben.
FILES "\VERKAUF"      'Wenn VERKAUF ein Verzeichnis ist,
                      'zeigt diese Anweisung alle
                      'Dateien in VERKAUF; wenn VERKAUF
                      'eine Datei im aktuellen
                      'Verzeichnis ist, zeigt diese
                      'Anweisung den Namen VERKAUF.
```

FIX-Funktion

Funktion

Gibt den abgeschnittenen ganzzahligen Teil von *x* an.

Syntax

FIX(*x*)

Anmerkungen

x ist ein numerischer Ausdruck. **FIX**(*x*) entspricht **SGN**(*x*)***INT**(**ABS**(*x*)). Der Unterschied zwischen **FIX** und **INT** besteht darin, daß **FIX** für negatives *x* die erste negative ganze Zahl größer als *x* angibt, während **INT** die erste negative ganze Zahl kleiner als *x* angibt.

Vergleichen Sie auch

CINT, INT

Beispiel

Die folgenden vier Anweisungen verdeutlichen den Unterschied zwischen **INT** und **FIX**:

```
PRINT INT(-99.8)
PRINT FIX(-99.8)
PRINT INT(-99.2)
PRINT FIX(-99.2)
```

Ausgabe

```
-100
-99
-100
-99
```

FOR...NEXT-Anweisung

Funktion

Führt eine Folge von Befehlen in einer Schleife aus. Die Anzahl der Schleifenwiederholungen ist vorgegeben.

Syntax

FOR *Zähler* = *Start* **TO** *Ende* [**STEP** *Schrittweite*]

.

.

.

NEXT [*Zähler* [, *Zähler*...]]

Anmerkungen

Die **FOR**-Anweisung hat folgende Argumente:

Argument	*Beschreibung*
Zähler	Eine numerische Variable, die als Schleifenzähler dient. Die Variable darf kein Verbund-Element oder Verbund-Datenfeld sein.
Start	Der Anfangswert des Zählers.
Ende	Der Endwert des Zählers.
Schrittweite	Der Betrag, um den der Zähler bei jedem Schleifendurchlauf erhöht wird.

Die Programmzeilen nach der Anweisung **FOR** werden so lange ausgeführt, bis die Anweisung **NEXT** erreicht wird. Anschließend wird *Zähler* um den von **STEP** angegebenen Betrag erhöht und mit dem endgültigen Wert *Ende* verglichen. (Falls Sie **STEP** nicht angeben, wird für die Schrittweite der Wert 1 angenommen.) Wenn *Zähler* immer noch nicht größer als *Ende* ist, verzweigt die Steuerung zu der Anweisung nach der Anweisung **FOR** zurück, und der Vorgang wird wiederholt. Ist *Zähler* größer als *Ende*, so wird die Ausführung mit der Anweisung nach der Anweisung **NEXT** fortgesetzt.

Wenn **STEP** negativ ist, wird der Zähler bei jedem Schleifendurchgang heruntergezählt und die Schleife ausgeführt, bis der Wert des Zählers kleiner als der Endwert ist.

Wenn der Startwert größer als der Endwert ist, wird die Schleife gar nicht ausgeführt. Die folgende Schleife wird nicht ausgeführt.

```
FOR I=3 TO 2
   PRINT EXP(2)
NEXT I
```

Ändern Sie den Wert einer Schleifenvariablen nicht innerhalb einer Schleife. Das Ändern des Wertes erschwert das Lesen und Debuggen des Programms.

Verschachtelte Schleifen

Sie können **FOR...NEXT**-Schleifen verschachteln, d. h. innerhalb einer **FOR...NEXT**-Schleife kann eine weitere **FOR...NEXT**-Schleife verwendet werden. Wenn mehrere Schleifen verschachtelt werden, muß jede Schleife einen eindeutigen Variablennamen als Zähler haben. Die **NEXT**-Anweisung für die innere Schleife muß vor der **NEXT**-Anweisung für die äußere Schleife stehen. Das folgende Beispiel ist richtig verschachtelt:

```
FOR I = 1 TO 10
   FOR J = 1 TO 10
      FOR K = 1 TO 10
      .
      .
      .
      NEXT K
   NEXT J
NEXT I
```

Eine **NEXT**-Anweisung mit der Form

```
NEXT K, J, I
```

entspricht der folgenden Anweisungsfolge:

```
NEXT K
NEXT J
NEXT I
```

EXIT FOR bietet eine sinnvolle Alternative, **FOR...NEXT**-Schleifen zu verlassen. Vergleichen Sie auch die Beschreibung für **EXIT FOR.**

Wichtig Wenn Sie in einer **NEXT**-Anweisung die Variable auslassen, entspricht die **NEXT**-Anweisung der aktuellsten **FOR**-Anweisung. Trifft das Programm auf eine **NEXT**-Anweisung vor der entsprechenden **FOR**-Anweisung, erscheint die Fehlermeldung `NEXT ohne FOR`.

Unterschiede zu BASICA

Anders als BASICA unterstützt QuickBASIC in seinen **FOR...NEXT**-Schleifen Steuerwerte (*Start, Ende* und *Zähler*) doppelter Genauigkeit. Falls die Steuerwerte jedoch in den Bereich der ganzen Zahlen fallen, sollten Sie auch ganzzahlige Steuerwerte benutzen, um eine größtmögliche Bearbeitungsgeschwindigkeit zu erreichen.

Beispiel

Das folgende Beispiel druckt die ersten 11 Spalten des Pascal'schen Dreiecks:

```
'Druckt die ersten MAXSPALT Spalten des
'Pascal'schen Dreiecks, in dem jede Zahl die
'Summe der Zahlen ist, die unmittelbar
'über und unter der vorhergehenden Spalte
'stehen:
CONST MAXSPALT=11
DIM A (MAXSPALT,MAXSPALT)
FOR M = 1 TO MAXSPALT
    A(M,1) = 1 : A(M,M) = 1   'Oberste und unterste
                              'Zahl in jeder Spalte
                              'ist 1
NEXT
FOR M = 3 TO MAXSPALT
   FOR N = 2 TO M-1
      A(M,N) = A(M-1,N-1) + A(M-1,N)
   NEXT
NEXT
Startzeile = 13         'Gehe in die Mitte des Bildschirms
FOR M = 1 to MAXSPALT
   Spalte = 6 * M
   Zeile = Startzeile
   FOR N = 1 TO M
      LOCATE Zeile,Spalte : PRINT A(M,N)
      Zeile = Zeile + 2   'Zum Druck der nächsten Zahl
                          '2 Zeilen tiefer gehen
   NEXT
   PRINT
   Startzeile = Startzeile - 1 'Nächste Spalte beginnt
                               '1 Zeile über der
                               'vorangegangenen Spalte
NEXT
```

Ausgabe

```
                                        1
                                    1
                                1       10
                            1       9
                        1       8       45
                    1       7       36
                1       6       28      120
            1       5       21      84
        1       4       15      56      210
    1       3       10      35      126
1       2       6       20      70      252
    1       3       10      35      126
        1       4       15      56      210
            1       5       21      84
                1       6       28      120
                    1       7       36
                        1       8       45
                            1       9
                                1       10
                                    1
                                        1
```

FRE-Funktion

Funktion

Gibt die Größe des zur Verfügung stehenden Speichers an.

Syntax 1

FRE (*Numerischer Ausdruck*)

Syntax 2

FRE (*Zeichenkettenausdruck*)

Anmerkungen

Mit einem numerischen Argument gibt die **FRE**-Funktion folgende Werte zurück:

Argument	*Zurückgegebener Wert*
-1	Die Größe des größten Nicht-Zeichenkettendatenfeldes, das dimensioniert werden kann, in Bytes.
-2	Der Betrag des ungenutzten Stapelplatzes, in Bytes, die dem Programm zur Verfügung stehen.
Jeder andere beliebige numerische Wert	Die Größe des nächsten freien Blocks des Zeichenkettenspeichers.

Wenn das Argument ein Zeichenkettenausdruck ist *(Zeichenkettenausdruck)*, gibt **FRE** die Größe des freien Zeichenkettenspeichers in Bytes an. Bevor **FRE** die Anzahl der freien Bytes angibt, komprimiert es den freien Zeichenkettenspeicher in einen einzigen Block.

Hinweis **FRE(-2)** gibt nur dann aussagefähige Werte aus, wenn das Programm ausgeführt wird. Werte, die durch **FRE(-2)** ausgegeben werden, sind nicht genau, wenn die Funktion von dem Befehlsfenster, während der Programm-Verfolgung oder der Anzeige einer Variablen, aufgerufen wird.

Beispiel

Das folgende Beispiel zeigt einige der Werte, die **FRE** vor und nach der Dimensionierung eines Datenfeldes zurückgibt:

```
' $DYNAMIC
PRINT "Vor Datenfeld-Dimensionierung: ";
PRINT FRE(""),FRE(0),FRE(-1)
DIM LARGE%(150,150), BIG$(5000)
PRINT "Nach Datenfeld-Dimensionierung: ";
PRINT FRE(""),FRE(0),FRE(-1)
```

Ausgabe

(Die von **FRE** angegebenen Werte können bei Ihrem Computer von diesem Beispiel abweichen.)

```
Vor Datenfeld-Dimensionierung: 58420   58420   322120
Nach Datenfeld-Dimensionierung: 38404   38404   276496
```

FREEFILE-Funktion

Funktion

Gibt die nächste freie BASIC-Dateinummer an.

Syntax

FREEFILE

Anmerkungen

Die Funktion **FREEFILE** gibt die nächste gültige, nicht benutzte Dateinummer an.

Sie können diese Funktion benutzen, um zu verhindern, daß **SUB**- oder **FUNCTION**-Prozeduren bereits vergebene Dateinummern benutzen.

Beispiel

Dieses Beispiel benutzt **FREEFILE**, um eine Dateinummer für eine zu öffnende Datei zu erhalten:

```
INPUT "Eingabe Dateiname: ",Dateiname$
DatNum = FREEFILE
OPEN Dateiname$ for INPUT as DatNum
PRINT Dateiname$;" Geöffnet als Datei # "; DatNum
```

Ausgabe

`Eingabe Dateiname:` **Data.dat**

```
Data.dat Geöffnet als Datei # 1
```

FUNCTION-Anweisung

Funktion

Deklariert Namen, Parameter und Code, die den Rumpf einer **FUNCTION**-Prozedur bilden.

Syntax

FUNCTION *Name* [(*Parameter-Liste*)][**STATIC**]
.
.
.
Name = *Ausdruck*
.
.
.
END FUNCTION

Anmerkungen

Die folgende Liste beschreibt die Teile der **FUNCTION**-Anweisung:

Teil	***Beschreibung***
Name	Der Name der Funktion. **FUNCTION**-Namen folgen denselben Regeln wie BASIC-Variablennamen und können ein Typdeklarationszeichen (%, &, !, # oder $) enthalten. Beachten Sie bitte, daß der Typ des Namens den Typ des Wertes, den die Funktion angibt, bestimmt. Wenn Sie zum Beispiel eine Funktion erstellen, die eine Zeichenkette angibt, müssen Sie ein Dollar-Zeichen an den Namen anhängen oder ihr einen Namen geben, den Sie als Zeichenketten-Namen mit einer **DEFSTR**-Anweisung definiert haben.
Parameter-Liste	Die von Kommata eingeschlossene Liste der Variablen, die **FUNCTION** übergeben werden. Die Parameter werden als Referenz übergeben, so daß jede Änderung eines Parameter-Wertes innerhalb der Funktion den Wert des Parameters im aufrufenden Programm ändert.
STATIC	Zeigt an, daß die lokalen Variablen der Funktion zwischen den Aufrufen gesichert werden sollen. Ohne **STATIC** werden die lokalen Variablen bei jedem Funktionsaufruf neu zugewiesen, wobei die Werte der Variablen verlorengehen, wenn die Kontrolle an das aufrufende Programm zurückgegeben wird. Das **STATIC**-Attribut hat keine Auswirkungen auf Variablen, die in einer **FUNCTION** verwendet werden und außerhalb der **FUNCTION** in **DIM**- oder **COMMON**-Anweisungen unter Benutzung des Schlüsselwortes **SHARED** deklariert sind.

Teil	***Beschreibung***
Ausdruck	Der Ausgabewert einer Funktion. Eine **FUNCTION** gibt einen Wert aus, indem der Wert dem Funktionsnamen zugewiesen wird. Wenn dem **FUNCTION**-Namen kein Wert zugewiesen ist, gibt **FUNCTION** einen voreingestellten Wert aus: Eine numerische Funktion gibt den Wert Null aus; eine Zeichenketten-Funktion gibt eine Null-Zeichenkette ("") aus.

Eine *Parameter-Liste* hat die folgende Syntax:

Variable [( )] [**AS** *Typ*][,*Variable* [( )][**AS** *Typ*]]

Eine *Variable* ist jede gültige BASIC-Variable. Der optionale *Typ* kann **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder ein benutzerdefinierter Typ sein.

Frühere BASIC-Versionen verlangten die Angabe der Anzahl der Dimensionen in Klammern nach einem Datenfeldnamen. Die Anzahl der Dimensionen wird nicht mehr benötigt. Nur die Klammern müssen angegeben werden, um anzuzeigen, daß der Parameter ein Datenfeld ist. Beispielsweise zeigt die folgende Anweisung, daß sowohl `Schluesselwoerter$` als auch `SchluesselwortTypen` Datenfelder sind:

```
FUNCTION AnalysiereZeile (Schluesselwoerter
$(), SchluesselwortTypen())
```

Eine **FUNCTION**-Prozedur ist ähnlich einer **SUB**-Prozedur: Sie akzeptiert Parameter, kann eine Serie von Anweisungen ausführen und kann den Wert ihrer Parameter verändern. Anders als eine **SUB** wird eine **FUNCTION** in einem Ausdruck in der gleichen Weise benutzt, wie eine eingebaute BASIC-Funktion.

Wie **SUB**-Prozeduren benutzen **FUNCTION**-Prozeduren lokale Variablen. Jede Variable, die sich nicht in der Parameter-Liste befindet, ist solange lokal zu der **FUNCTION**, sofern sie nicht in einer **SHARED**-Anweisung deklariert ist oder in einer **DIM**- oder **COMMON**-Anweisung mit dem Attribut **SHARED** auftritt.

Um einen Wert für eine Funktion zu erhalten, muß der Wert dem Namen der Funktion zugewiesen werden. In einer Funktion mit dem Namen `BinaerSuche` können Sie dem Namen zum Beispiel den Wert der Konstanten `FALSE` zuweisen um anzuzeigen, daß der gesuchte Wert nicht gefunden wurde:

```
FUNCTION BinaerSuche(...)
CONST FALSE=0
.
.
.
   'Wert nicht gefunden. Gib den Wert von FALSE aus.
```

```
    IF unten > oben THEN
    BinaerSuche=FALSE
    EXIT FUNCTION
    END IF
.
.
.
END FUNCTION
```

Die Verwendung des **STATIC**-Schlüsselwortes erhöht etwas die Ausführungsgeschwindigkeit. **STATIC** wird normalerweise nicht mit rekursiven **FUNCTION**-Prozeduren benutzt. Sehen Sie hierzu das Beispiel unten.

Die **EXIT FUNCTION**-Anweisung bietet ein alternatives Verlassen einer **FUNCTION**. Sehen Sie hierzu die Beschreibung zu der Anweisung **EXIT**.

Weil BASIC teilweise arithmetische Ausdrücke zur besseren Effizienz neu zusammenstellt, vermeiden Sie den Gebrauch von **FUNCTION**-Prozeduren, die Programmvariablen in arithmetischen Ausdrücken verändern. Ebenso vermeiden Sie in E/A-Anweisungen die Benutzung von **FUNCTION**-Prozeduren, die E/A-Operationen ausführen. Die Benutzung von **FUNCTION**-Prozeduren, die grafische Operationen ausführen, können ebenso zu Seiteneffekten führen.

QuickBASIC-**FUNCTION**-Prozeduren sind rekursiv – sie können sich selbst aufrufen, um eine bestimmte Aufgabe zu erfüllen. Sehen Sie hierzu das zweite Beispiel unten und Kapitel 4, "Programme und Module".

Vergleichen Sie auch

DECLARE (BASIC), DEF FN, EXIT, STATIC, SUB

Beispiele

Das folgende Beispiel benutzt eine Funktion, um die Anzahl der Selbstlaute in einer Zeichenkette zu zählen:

```
' Funktionsdefinition
FUNCTION AnzVokal (A$) STATIC
   Anz = 0
   ' Gehe Zeichen für Zeichen durch A$.
   FOR I = 1 TO LEN (A$)
      C$ = UCASE$ (MID$(A$,I,1))
      IF INSTR ("AEIOU",C$)<> 0 THEN
         ' Vokal gefunden--zähle ihn
         Anz = Anz + 1
      END IF
   NEXT I
```

```
   AnzVokal = Anz
END FUNCTION
A$ = "Wie langweilig ist das Leben ohne Computer."
PRINT CHR$(34)+A$+CHR$(34)
PRINT "Die Anzahl der Vokale in der "
PRINT "Zeichenkette ist :";AnzVokal (A$)
```

Ausgabe

```
"Wie langweilig ist das Leben ohne Computer."
Die Anzahl der Vokale in der Zeichenkette ist : 15
```

Das folgende Beispiel benutzt eine rekursive Funktion (eine Funktion, die sich selbst aufruft), um die Länge einer Zeichenkette festzustellen. Beachten Sie, daß das **STATIC**-Schlüsselwort nicht benutzt wurde.

```
FUNCTION ZkLaenge(X$)
   IF X$ = "" THEN
      ' Die Länge einer Null-Zeichenkette ist Null.
      ZkLaenge=0
   ELSE
      ' Nicht-Null-Zeichenkette--rekursiver Aufruf
      ' Die Länge einer Nicht-Null-Zeichenkette ist 1
      ' plus der Länge des Restes der Zeichenkette.
      ZkLaenge=1+ZkLaenge(MID$(X$,2))
   END IF
END FUNCTION
PRINT "Bitte Zeichenkette eingeben ";
LINE INPUT ":",EinZeikette$
PRINT "Die Länge der Zeichenkette ist";
 PRINT ZkLaenge(EinZeikette$)
```

Ausgabe

```
Bitte Zeichenkette eingeben: Sommer Sonne Strand
Die Länge der Zeichenkette ist 19
```

GET-Anweisung – Datei-E/A

Funktion

Liest von einer Disketten-/Festplattendatei in eine Variable oder einen Puffer für Direktzugriff ein.

Syntax

GET [#] *Dateinummer* [,[*Satznummer*][,*Variable*]]

Anmerkung

Die folgende Liste beschreibt die Argumente der **GET**-Anweisung:

Argument	***Beschreibung***
Dateinummer	Die in der **OPEN**-Anweisung verwendete Nummer zum Öffnen der Datei.
Satznummer	Die Nummer des zu lesenden Satzes für Direktzugriffsdateien. Für Dateien im Binärmodus die Byte-Position in der Datei, an der das Lesen beginnt. Der erste Satz oder die erste Byte-Position in einer Datei ist 1. Wenn Sie *Satznummer* nicht angeben, wird der nächste Satz oder das nächste Byte (der/das nach dem letzten **GET** oder **PUT**, oder der/das, auf den/das das letzte **SEEK** zeigt) in den Puffer gelesen. Die größtmögliche Satznummer ist 2^{31}-1 oder 2.147.483.647.
Variable	Die Variable zum Aufnehmen der Eingabe aus der Datei. Wenn Sie eine Variable benutzen, ist es nicht notwendig, mit **CVD, CVL**, **CVI** oder **CVS** Satzfelder in Zahlen umzuwandeln. Sie dürfen für die Datei keine **FIELD**-Anweisung verwenden, wenn Sie das Argument *Variable* benutzen. Für Direktzugriffsdateien können Sie jede Variable verwenden, solange die Länge dieser Variablen kleiner oder gleich der Länge des Satzes ist. Normalerweise wird eine passend zu den Feldern in einem Datensatz definierte Verbundvariable verwendet. Für Dateien im Binärmodus können Sie jede beliebige Variable verwenden. Die Anweisung **GET** liest soviele Bytes, wie sich in der Variablen befinden.

Argument	***Beschreibung***
	Wenn Sie Zeichenkettenvariablen mit variabler Länge verwenden, liest die Anweisung soviele Bytes, wie Zeichen in der Zeichenkette vorkommen. Zum Beispiel lesen die folgenden zwei Anweisungen 10 Bytes aus Datei Nummer 1:

```
VarZeich$=STRING$ (10, " ")
GET #1,,VarZeich$
```

Weitere Informationen über die Verwendung von Variablen anstelle von **FIELD**-Anweisungen für Direktzugriffsdateien finden Sie in den Beispielen und in Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Ein Satz kann nicht länger als 32.767 Bytes sein.

Sie können die *Satznummer*, die *Variable* oder beides weglassen. Wenn Sie nur die *Satznummer* weglassen, müssen Sie die Kommata weiterhin angeben:

```
GET #4,,DateiPuffer
```

Wenn Sie beide Argumente weglassen, müssen Sie die Kommata nicht angeben:

```
GET #4
```

Die Anweisungen **GET** und **PUT** erlauben Eingaben und Ausgaben fester Länge für BASIC-Kommunikationsdateien. Verwenden Sie **GET** mit Vorsicht, weil **GET** bei Übertragungsfehlern unbegrenzt auf *Satznummer*-Zeichen wartet.

Hinweis Wenn Sie **GET** mit der Anweisung **FIELD** verwenden, können Sie **INPUT #** oder **LINE INPUT #** nach einer **GET**-Anweisung benutzen, um Zeichen aus dem Puffer der Direktzugriffsdatei zu lesen.

Nach einer **GET**-Anweisung können Sie mit der Funktion **EOF** prüfen, ob dieses **GET** hinter dem Dateiende-Zeichen zu lesen versucht hat.

Vergleichen Sie auch

CVI, CVS, CVL, CVD, FIELD; INPUT #, LINE INPUT #, LSET, MKD$, MKI$, MKL$, MKS$, PUT (Datei-E/A), RSET, TYPE

Beispiel

Das nachstehende Programm öffnet die Datei `TESTDAT.DAT` für Direktzugriff und zeigt den Inhalt auf dem Bildschirm an:

```
' Lies und zeige den Inhalt einer Datei, die
' einen Namen mit bis zu 20 Zeichen und ein
' Testergebnis enthält.
' Definiere Verbund-Datenfelder.
TYPE TestVerbund
   NameFeld  AS STRING*20
   ErgebFeld AS SINGLE
END TYPE
' Öffne die Testdaten-Datei.
DIM DateiPuffer AS TestVerbund
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 LEN=LEN(DateiPuffer)
' Berechne die Anzahl der Sätze in der Datei.
Max=LOF(1)/LEN(DateiPuffer)
' Lies und schreibe den Inhalt jedes
' Verbundes.
FOR I=1 TO Max
   GET #1,I,DateiPuffer
   PRINT DateiPuffer.NameFeld, DateiPuffer.ErgebFeld
NEXT I
CLOSE #1
```

GET-Anweisung – Grafik

Funktion

Speichert Grafiken vom Bildschirm.

Syntax

GET [**STEP**](*x1,y1*)-[**STEP**](*x2,y2*), *Datenfeldname* [(*Indizes*)]

Anmerkungen

Die folgende Liste beschreibt die Teile der **GET**-Anweisung:

Teil	*Beschreibung*
x1,y1,x2,y2	Koordinaten, die eine rechteckige Fläche auf dem Bildschirm markieren. Die Platzhalter *x1*, *y1*, *x2* und *y2* sind numerische Ausdrücke, die die Koordinaten sich diagonal gegenüberliegender Ecken der rechteckigen Fläche angeben.
STEP	Dieses Schlüsselwort zeigt an, daß Koordinaten relativ zum letzten gezeichneten Punkt sind. Wenn zum Beispiel der letzte gezeichnete Punkt (10,10) war, ist die tatsächliche Koordinate, auf die durch **STEP** (5,10) Bezug genommen wird, (5+10,10+10) oder (15,20). Wenn das zweite Koordinatenpaar in der **GET**-Anweisung ein **STEP**-Argument hat, ist dies relativ zu dem ersten Koordinatenpaar in der Anweisung.
Datenfeldname	Der dem Datenfeld zugewiesene Name, welches das Bild enthält. Dieses Feld kann jeden numerischen Typ haben; seine Dimensionen müssen genügend groß sein, um das gesamte Bild aufnehmen zu können.
Indizes	Numerische Konstanten oder Variablen, die das Element des Datenfeldes anzeigen, mit dem das gespeicherte Bild beginnt.

Die **GET**-Anweisung überträgt einen Bildschirmausschnitt in das durch *Datenfeldname* angegebene Datenfeld. Die Anweisung **PUT**, das Gegenstück von **GET**, überträgt das im Datenfeld gespeicherte Bild auf den Bildschirm.

Die erforderliche Größe des Datenfeldes in Bytes kann man mit folgender Formel berechnen:

4+INT(((*x2-x1*+**1**)*(*Bits pro Bildpunkt pro Ebene*)+**7**)/**8**)* *Ebene**((*y2-y1*)+**1**)

Der Wert *Bits pro Bildpunkt pro Ebene* hängt von der in der **SCREEN**-Anweisung gesetzten Spezifikation ab. Die folgende Tabelle zeigt die Anzahl der Bits pro Bildpunkt pro Ebene und Anzahl der Ebenen für jeden Bildschirmmodus.

Modus	*Bits pro Bildpunkt pro Ebene*	*Ebenen*
SCREEN 1	2	1
SCREEN 2	1	1
SCREEN 7	1	4
SCREEN 8	1	4
SCREEN 9	1	2 (wenn 64K EGA-Speicher vorhanden) 4 (wenn mehr als 64K EGA-Speicher vorhanden)
SCREEN 10	1	2
SCREEN 11	1	1
SCREEN 12	1	4
SCREEN 13	8	1

Pro Datenfeldelement werden benötigt:

- Zwei Bytes für ein ganzzahliges Datenfeldelement
- Vier Bytes für ein 4-Byte-Ganzzahl-Datenfeldelement
- Vier Bytes für ein Datenfeldelement einfacher Genauigkeit
- Acht Bytes für ein Datenfeldelement doppelter Genauigkeit

Angenommen, Sie wollen die Anweisung **GET** benutzen, um ein Bild mit hoher Auflösung (**SCREEN 2**) zu speichern. Wenn die Koordinaten der oberen linken Ecke des Bildes (0,0) und die Koordinaten der unteren rechten Ecke (32,32) sind, ist die erforderliche Größe des Datenfeldes, in Bytes, 4+INT((33*1+7)/8)*1*(33) oder 169. Dies bedeutet, daß ein ganzzahliges Datenfeld mit 85 Elementen zur Aufnahme des Bildes ausreicht.

Außer bei einem ganzzahligen oder 4-Byte-ganzzahligen Datenfeld erscheint der Inhalt eines Datenfeldes nach einer **GET**-Anweisung bei sofortiger Überprüfung bedeutungslos. Die Durchsicht oder Manipulation von Nichtganzzahl-Datenfeldern, die Grafiken enthalten, kann zu einem Laufzeitfehler führen.

GET und **PUT** können auch zur Erzeugung von Animationen benutzt werden. Weitere Informationen finden Sie in Kapitel 5, "Grafiken", in *Programmieren in BASIC: Ausgewählte Themen.*

Vergleichen Sie auch

PUT (Grafik)

Beispiel

Siehe Beispiel für **BSAVE.**

GOSUB...RETURN-Anweisungen

Funktion

Verzweigt zu einer Unterroutine und kehrt von dort in die aufrufende Routine zurück.

Syntax

GOSUB {*Zeilenmarke1* | *Zeilennummer1*}

.

.

.

RETURN [*Zeilenmarke2* | *Zeilennummer2*]

Anmerkungen

Die **GOSUB...RETURN**-Anweisungen haben folgende Argumente:

Argument	***Beschreibung***
Zeilenmarke1, Zeilennummer1	Die Zeilennummer oder Zeilenmarke, die die erste Zeile der Unterroutine ist.
Zeilenmarke2, Zeilennummer2	Die Zeilenmarke oder Zeilennummer, zu der die Unterroutine zurückkehrt.

Hinweis BASICs **SUB**- und **FUNCTION**-Prozeduren bieten eine besser strukturierte Alternative zu **GOSUB...RETURN**-Unterroutinen.

Zusätzlich zu **RETURN** ohne Argument unterstützt BASIC die Anweisung **RETURN** mit einer Zeilenmarke oder einer Zeilennummer, die es erlaubt, einen Rücksprung aus der Unterroutine zu der Anweisung mit der angegebenen Zeilennummer oder -marke auszuführen, statt des Rücksprungs in die Anweisung nach der **GOSUB**-Anweisung. Benutzen Sie diesen Typ des Rücksprungs mit Vorsicht.

Sie können eine Unterroutine in einem Programm beliebig oft aufrufen. Außerdem können Sie eine Unterroutine aus einer anderen Unterroutine heraus aufrufen. Wie tief Sie Unterroutinen verschachteln können, ist nur durch den zur Verfügung stehenden Stapelbereich begrenzt (Sie können den Stapelbereich mit der Anweisung **CLEAR** vergrößern). Unterroutinen, die sich selbst aufrufen (rekursive Unterroutinen), können zum Überlauf des Stapelbereichs führen. **RETURN** mit einer Zeilenmarke oder einer Zeilennummer kann die Kontrolle nur an eine Anweisung im Hauptprogramm zurückgeben. Siehe hierzu das Programmbeispiel.

Eine Unterroutine kann mehr als eine **RETURN**-Anweisung enthalten. Einfache **RETURN**-Anweisungen (ohne die Option *Zeilenmarke2,Zeilennummer2*) in einer Unterroutine veranlassen BASIC, zu der Anweisung nach der zuletzt ausgeführten **GOSUB**-Anweisung zurückzukehren.

Unterroutinen können an jeder Stelle des Programms stehen, doch sollte man sie deutlich erkennbar vom Hauptprogramm absetzen. Damit Sie nicht versehentlich in eine Unterroutine gelangen, steuern Sie das Programm durch eine vorangestellte **STOP**-, **END**- oder **GOTO**-Anweisung um die Unterroutine herum.

Warnung Die vorstehenden Informationen über Unterroutinen gelten nur für die Ziele von **GOSUB**-Anweisungen und *nicht* für Unterprogramme, die durch die **SUB**-Anweisungen begrenzt werden. Das Aufrufen und Verlassen von **SUB**-Blöcken mittels **GOSUB...RETURN**-Anweisungen wird nicht unterstützt.

Vergleichen Sie auch

RETURN, SUB

Beispiel

Das nachstehende Beispiel zeigt den richtigen Gebrauch der Anweisung **RETURN** *Zeilenmarke*:

```
PRINT "im Modul-Ebenen-Code"
GOSUB Sub1
PRINT "diese Zeile in der Hauptroutine sollte";
PRINT " übersprungen werden"
Marke1:
   PRINT "zurück im Modul-Ebenen-Code"
   END
```

```
Sub1:
   PRINT "in Unterroutine 1"
   GOSUB Sub2
   PRINT "diese Zeile in Unterroutine 1 ";
   PRINT "sollte übersprungen werden"
Marke2:
   PRINT "zurück in Unterroutine 1"
   RETURN Marke1

Sub2:
   PRINT "in Unterroutine 2"
   RETURN Marke2    'Kann nicht von hier zum
                    'Hauptprogramm zurückkehren - nur zu
                    'SUB1
```

Ausgabe

```
im Modul-Ebenen-Code
in Unterroutine 1
in Unterroutine 2
zurück in Unterroutine 1
zurück im Modul-Ebenen-Code
```

GOTO-Anweisung

Funktion

Unbedingte Verzweigung zu der angegebenen Zeile.

Syntax

GOTO {*Zeilenmarke* | *Zeilennummer*}

Anmerkungen

Die **GOTO**-Anweisung bietet eine Möglichkeit, ohne Bedingung zu einer anderen Zeile (*Zeilenmarke* oder *Zeilennummer*) zu verzweigen. Eine **GOTO**-Anweisung kann nur zu einer anderen Anweisung auf derselben Programmebene verzweigen. Sie können **GOTO** nicht benutzen, um in eine **SUB FUNCTION** oder mehrzeilige **DEF FN**-Funktion zu verzweigen oder eine solche zu verlassen. Sie können jedoch mit **GOTO** den Programmfluß innerhalb all dieser Strukturen steuern.

Es ist ratsam, strukturierte Schleifenanweisungen (**DO...LOOP, FOR, IF...THEN...ELSE, SELECT CASE**) anstelle von **GOTO**-Anweisungen zum Verzweigen zu benutzen, weil ein Programm mit vielen **GOTO**-Anweisungen schwierig zu lesen und auszutesten ist.

Beispiel

Das folgende Programm gibt die Fläche des Kreises mit dem eingegebenen Radius aus:

```
PRINT "Eingabe 0 zum Beenden."
Start:
   INPUT R
   IF R <= 0 THEN
      END
   ELSE
      A = 3.14 * R^2
      PRINT "Fläche =";A
   END IF
GOTO Start
```

Ausgabe

```
Eingabe 0 zum Beenden.
? 5
Fläche = 78.5
```

HEX$-Funktion

Funktion

Gibt eine Zeichenkette an, die den Hexadezimalwert des dezimalen Arguments *Ausdruck* darstellt.

Syntax

HEX$ (*Ausdruck*)

Anmerkungen

Das Argument *Ausdruck* wird auf eine ganze Zahl oder, falls der *Ausdruck* außerhalb des Ganzzahlbereiches liegt, auf eine lange Ganzzahl gerundet, bevor **HEX$** es auswertet.

Vergleichen Sie auch

OCT$

Beispiel

Das folgende Beispiel gibt die hexadezimale Repräsentation eines einzugebenden Wertes aus:

```
INPUT X
A$ = HEX$(X)
PRINT X "dezimal ist" A$ "hexadezimal"
```

Ausgabe

```
? 32
32 dezimal ist 20 hexadezimal
```

IF...THEN...ELSE-Anweisung

Funktion

Ermöglicht die bedingte Ausführung oder Verzweigung, die auf der Auswertung eines Booleschen Ausdrucks beruht.

Syntax 1 (Einzeilig)

IF *Boolescher Ausdruck* **THEN** *Dann-Teil* [**ELSE** *Sonst-Teil*]

Syntax 2 (Block)

IF *Boolescher Ausdruck1* **THEN**
  [*Anweisungsblock-1*]
[**ELSEIF** *Boolescher Ausdruck2* **THEN**
  [*Anweisungsblock-2*]]
.
.
.
[**ELSE**
  [*Anweisungsblock-n*]]
END IF

Anmerkungen

Die einzeilige Form der Anweisung eignet sich am besten für kurze, geradlinige Tests, in denen nur eine Aktion ausgeführt wird.

Die Block-Form bietet mehrere Vorteile:

- Die Block-Form bietet mehr Struktur und Flexibilität als die einzeilige Form, indem die Möglichkeit zur bedingten Verzweigung über mehrere Zeilen besteht.
- Mit der Block-Form können mehrere komplexe Bedingungen geprüft werden.
- Die Block-Form ermöglicht die Anwendung langer Anweisungen und Strukturen innerhalb des **THEN...ELSE**-Teils der Anweisung.
- Mit der Block-Form wird die Programmstruktur von der Logik bestimmt und nicht dadurch, wieviele Anweisungen auf eine Zeile passen.

Programme, die die Block-Form **IF...THEN...ELSE** benutzen, sind zumeist leichter zu lesen, zu pflegen und zu debuggen.

Die einzeilige Form ist nicht zwingend. Jedes Programm, das die einzeilige **IF...THEN...ELSE**-Anweisung benutzt, kann in der Block-Form geschrieben werden.

Einzeiliges IF...THEN...ELSE

Die folgende Liste beschreibt die Teile der einzeiligen Form:

Teil	***Beschreibung***
Boolescher Ausdruck	Jeder als wahr (nicht Null) oder falsch (Null) auswertbare Ausdruck.
Dann-Teil Sonst-Teil	Die Anweisungen oder Verzweigungen, die ausgeführt werden, wenn *Boolescher Ausdruck* wahr (*Dann-Teil*) oder falsch (*Sonst-Teil*) ist. Beide Teile haben dieselbe nachfolgend beschriebene Syntax.

Sowohl der *Dann-Teil* als auch der *Sonst-Teil* haben die folgende Syntax:

{*Anweisungen* | [**GOTO**]*Zeilennummer* | **GOTO** *Zeilenmarke*}

Die folgende Liste beschreibt die Teile der *Dann-Teil*- und *Sonst-Teil*-Syntax.

Teil	***Beschreibung***
Anweisungen	Eine oder mehrere durch Doppelpunkte getrennte BASIC-Anweisung/en.
Zeilennummer	Eine gültige BASIC-Programm-Zeilennummer.
Zeilenmarke	Eine gültige BASIC-Zeilenmarke.

Beachten Sie bitte, daß **GOTO** bei einer Zeilennummer wahlfrei ist, während es bei einer Zeilenmarke angegeben werden muß.

Der *Dann-Teil* wird ausgeführt, wenn *Boolescher Ausdruck* wahr ist. Wenn *Boolescher Ausdruck* falsch ist, wird der *Sonst-Teil* ausgeführt. Wenn die Klausel **ELSE** nicht vorhanden ist, geht die Kontrolle an die nächste Programmanweisung über.

Es können mehrere Anweisungen mit einer Bedingung verknüpft sein, aber sie müssen alle auf derselben Zeile stehen und durch Doppelpunkte getrennt sein:

```
IF A > 10 THEN A=A+1:B=B+A:LOCATE 10,22:PRINT B,A
```

Block-IF...THEN...ELSE

Die folgende Liste beschreibt die Teile des Block-**IF...THEN...ELSE**:

Teil	*Beschreibung*
Boolescher Ausdruck1, Boolescher Ausdruck2	Jeder als wahr (nicht Null) oder falsch (Null) auswertbare Ausdruck.
Anweisungsblock-1, Anweisungsblock-2, Anweisungsblock-n	Eine oder mehrere BASIC-Anweisung/en in einer oder mehreren Zeile/n.

QuickBASIC führt ein Block-**IF** aus, indem der erste Boolesche Ausdruck (*Boolescher Ausdruck1*) getestet wird. Wenn der Boolesche Ausdruck wahr ist (Nicht-Null), werden die Anweisungen ausgeführt, die auf **THEN** folgen. Falls der erste Boolesche Ausdruck falsch ist (Null), beginnt QuickBASIC mit der Auswertung jeder folgenden **ELSEIF**-Bedingung. Wenn QuickBASIC eine wahre Bedingung vorfindet, werden die Anweisungen, die hinter dem dazugehörigen **THEN** stehen, ausgeführt. Wenn keine der **ELSEIF**-Bedingungen wahr sind, werden die hinter **ELSE** stehenden Anweisungen ausgeführt. Nachdem die Anweisungen, die hinter einem **THEN** oder **ELSE** stehen, ausgeführt sind, fährt das Programm mit der Anweisung fort, die hinter dem **END IF** steht.

Die **ELSE**- und **ELSEIF**-Blöcke sind beide wahlfrei.

Sie können beliebig viele **ELSEIF**-Klauseln in einem **IF**-Block verwenden.

Jeder der Anweisungsblöcke kann eingefügte Block-**IF**-Anweisungen enthalten.

QuickBASIC orientiert sich an dem, was dem **THEN**-Schlüsselwort folgt, um festzustellen, ob eine **IF**-Anweisung ein Block-**IF** ist oder nicht. Wenn der **THEN**-Anweisung etwas anderes als ein Kommentar folgt, wird die Anweisung als einzeilige **IF**-Anweisung behandelt.

Eine Block-**IF**-Anweisung muß die erste Anweisung in einer Zeile sein. Vor den Anweisungsteilen **ELSE, ELSEIF** und **END IF** dürfen nur Zeilennummern oder Zeilenmarken stehen.

Der Block *muß* mit der Anweisung **END IF** beendet werden.

Vergleichen Sie auch

SELECT CASE und Kapitel 1, "Strukturen zur Ablaufsteuerung", in *Programmieren in BASIC: Ausgewählte Themen.*

Beispiele

Die folgenden Programmfragmente zeigen die Benutzung und den Unterschied der einzeiligen und der Block-**IF...THEN...ELSE**-Anweisung.

Das erste Beispiel zeigt die einzeilige **IF...THEN...ELSE**-Form.

```
DO
PRINT "Eine Zahl eingeben, die größer als 0"
PRINT "und kleiner als 10000 ist";
INPUT ":", X
IF X>=0 AND X<10000 THEN EXIT DO ELSE PRINT _
X;"außerhalb des Bereichs"
LOOP
IF X<10 THEN Y=1 ELSE IF X<100 THEN Y=2 ELSE _
IF X<1000 THEN Y=3 ELSE Y=4
PRINT "Die Zahl hat";Y;" Zeichen"
```

Im zweiten Beispiel wird die Block-**IF...THEN...ELSE**-Anweisung verwendet, um das Beispiel lesbarer und leistungsfähiger zu machen:

```
DO
PRINT "Eine Zahl eingeben, die größer als 0"
PRINT "und kleiner als 10000 ist";
   INPUT ":", X
   IF X>=0 AND X<10000 THEN
      EXIT DO
   ELSE
      PRINT X;"außerhalb des Bereichs"
   END IF
LOOP
IF X<10 THEN
   y=1
ELSEIF X<100 THEN
   Y=2
ELSEIF X<1000 THEN
   Y=3
ELSEIF X<10000 THEN
   Y=4
ELSE
   Y=5
ENDIF
PRINT "Die Zahl hat";Y;" Zeichen"
```

INKEY$-Funktion

Funktion

Liest ein Zeichen von der Tastatur.

Syntax

INKEY$

Anmerkungen

INKEY$ gibt eine Ein- oder Zwei-Byte-Zeichenkette an, die ein vom Standard-Eingabegerät eingelesenes Zeichen enthält, oder eine Null-Zeichenkette, wenn dort keine Zeichen anstehen. Eine aus einem Zeichen bestehende Zeichenkette enthält das eigentliche Zeichen von der Tastatur, während eine aus zwei Zeichen bestehende Zeichenkette einen erweiterten Code angibt, dessen erstes Zeichen hexadezimal 00 ist. Eine komplette Liste dieser Codes finden Sie in Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes".

Das Standard-Eingabegerät ist in der Regel die Tastatur. **INKEY$** gibt keine Zeichen auf den Bildschirm aus; statt dessen übergibt die Funktion alle Zeichen, bis auf die folgenden, an das Programm:

- STRG+UNTBR beendet das Programm
- STRG+ALT+ENTF bootet ein System neu
- STRG+NUM-FESTSTELLTASTE hält die Programmausführung an
- DRUCK druckt den Bildschirminhalt aus

Ein allein ausführbares **.EXE**-Programm kann jedoch ein STRG+UNTBR lesen, ohne dadurch das Programm zu beenden, wenn die Option Debug oder **/d** während des Kompilierens nicht angegeben wird.

Beispiel

Der folgende Programmausschnitt zeigt einen üblichen Gebrauch von **INKEY$**: Anhalten, bis der Benutzer eine Taste betätigt.

```
PRINT "Weiter mit jeder Taste..."
DO
LOOP WHILE INKEY$=""
```

INP-Funktion

Funktion

Gibt das von einem E/A-Anschluß eingelesene Byte an.

Syntax

INP (*Anschluß*)

Anmerkungen

Anschluß muß eine ganze Zahl im Bereich von 0 bis 65.535 sein. Die Funktion **INP** ergänzt die Anweisung **OUT**.

Die Anweisungen **INP** und **OUT** ermöglichen einem BASIC-Programm die direkte Kontrolle über die Hardware eines Systems, mit Hilfe der E/A-Anschlüsse. Diese Befehle sollten vorsichtig benutzt werden, weil sie direkt die Systemhardware manipulieren.

Vergleichen Sie auch

OUT, WAIT

Beispiel

Siehe Beispiel für die Anweisung **OUT**.

INPUT-Anweisung

Funktion

Ermöglicht die Tastatureingabe während der Programmausführung.

Syntax

INPUT [;] ["*Anfrage*"{; | ,}] *Variablenliste*

Anmerkungen

Die folgende Liste beschreibt die Teile der **INPUT**-Anweisung:

Teil	*Beschreibung*
;	Ein Semikolon direkt hinter **INPUT** hält den Cursor auf derselben Zeile, nachdem der Benutzer die EINGABETASTE betätigt hat.
Anfrage	Eine Zeichenkettenkonstante oder Variable, die vor der Eingabeaufforderung geschrieben wird.
;	Am Ende von *Anfrage* wird ein Fragezeichen gedruckt.
,	Unterdrückt das Drucken des Fragezeichens nach *Anfrage*.
Variablenliste	Eine durch Kommata getrennte Liste von Variablen, die die Eingabewerte aufnehmen. Vergleichen Sie auch die nachfolgenden Erläuterungen.

INPUT bewirkt, daß das Programm anhält und mit einem Fragezeichen anzeigt, daß es auf die Eingabe von Daten wartet. Anschließend können Sie die erforderlichen Daten über die Tastatur eingeben.

Die eingegebenen Daten werden den Variablen in *Variablenliste* zugewiesen. Die Anzahl der eingegebenen Daten muß mit der Anzahl der Variablen in der Liste identisch sein. Das erste Zeichen nach einem Komma, das weder ein Leerzeichen noch ein Wagenrücklauf oder Zeilenvorschub ist, wird als Anfang eines neuen Datenelementes angenommen.

Die Variablennamen in der Liste können numerische oder Zeichenketten-Variablennamen (einschließlich indizierter Variablen), Datenfeldelemente oder Verbundelemente sein. Der Typ jedes eingegebenen Datenfeldelementes muß mit dem Variablentyp übereinstimmen. (Zeichenketten, die zu einer **INPUT**-Anweisung eingegeben werden, müssen nicht in Anführungszeichen gesetzt werden.) Wenn dieses erste Zeichen ein Anführungszeichen (") ist, besteht die Zeichenkette aus allen Zeichen, die zwischen dem ersten und dem zweiten Anführungszeichen eingelesen werden. Dies bedeutet, daß eine Zeichenkette in Anführungszeichen kein Anführungszeichen als Zeichen enthalten darf. Ist das erste Zeichen der Zeichenkette kein Anführungszeichen, so ist diese eine Zeichenkette ohne Anführungszeichen und wird mit einem Komma, Wagenrücklauf oder Zeilenvorschub abgeschlossen.

Eingaben, die in Verbundelementen gespeichert werden, müssen als Einzelelemente eingegeben werden:

```
TYPE Demograph
   Name  AS STRING*25
   Alter AS INTEGER
END TYPE
DIM Person AS Demograph
PRINT "Geben Sie Namen und Alter ein ";
INPUT ":"; Person.Name,Person.Alter
```

Die Beantwortung einer **INPUT**-Anweisung mit zuvielen oder zuwenigen Eingaben oder mit dem falschen Werttyp (z. B. numerischer Typ statt Zeichenkettentyp) führt zu folgender Fehlermeldung:

```
Korrigieren Sie die Eingabe
```

Die eingegebenen Werte werden so lange nicht zugewiesen, bis Sie eine zulässige Antwort geben.

Es besteht die Möglichkeit, die Eingabezeile zu editieren, bevor sie EINGABE drücken. In der nachstehenden Liste werden die Tastenkombinationen beschrieben, mit denen Sie den Cursor bewegen, Text löschen und einfügen können:

Tasten	*Funktion*
STRG+\ oder →	Bewegt den Cursor ein Zeichen nach rechts.
STRG+] oder ←	Bewegt den Cursor ein Zeichen nach links.
STRG+F oder STRG+→	Bewegt den Cursor ein Wort nach rechts.
STRG+B oder STRG+←	Bewegt den Cursor ein Wort nach links.
STRG+K oder POS1	Bewegt den Cursor an den Anfang der Eingabezeile.
STRG+N oder ENDE	Bewegt den Cursor an das Ende der Eingabezeile.

Tasten	***Funktion***
STRG+R oder EINFG	Schaltet den Einfügemodus ein und aus. Wenn der Einfügemodus eingeschaltet ist, werden die Zeichen in der aktuellen Cursorposition und rechts von ihr bei der Eingabe von neuen Zeichen nach rechts geschoben.
STRG+I oder TAB	Bewegt den Cursor zu den TAB-Positionen rechts und fügt Zeichen ein (Einfügemodus eingeschaltet) oder überschreibt sie (Einfügemodus ausgeschaltet).
ENTF	Löscht das Zeichen in der aktuellen Cursorposition.
STRG+H oder RÜCKTASTE	Löscht das Zeichen links vom Cursor, es sei denn, der Cursor steht am Anfang der Eingabe; in diesem Fall wird das Zeichen in der aktuellen Cursorposition gelöscht.
STRG+E oder STRG+ENDE	Löscht bis zum Zeilenende.
STRG+U oder ESC	Löscht die gesamte Zeile, ohne Rücksicht auf die Cursorposition.
STRG+M oder EINGABETASTE	Speichert die eingegebene Zeile.
STRG+T	Schaltet die Funktionstasten-Belegungsanzeige unten auf dem Bildschirm ein und aus.
STRG+UNTBR oder STRG+C	Schließt die Eingabe ab, beendet das Programm.

Beispiel

Das folgende Beispiel berechnet die Fläche eines Kreises von einem einzugebenden Radius:

```
PI = 3.141593 : R = -1
PRINT "Gib den Radius ein (0 zum Beenden):"
DO WHILE R
   PRINT
   INPUT;"Wenn Radius = ", R
   IF R > 0 THEN
       A = PI*R^2
       PRINT ", ist die Fläche des Kreises =", A
   END IF
LOOP
```

Ausgabe

```
Gib den Radius ein (0 zum Beenden):
Wenn Radius = 3, ist die Fläche des Kreises = 28.27434
Wenn Radius = 4, ist die Fläche des Kreises = 50.26549
Wenn Radius = 0
```

INPUT #-Anweisung

Funktion

Liest Daten aus einem sequentiellen Gerät oder einer sequentiellen Datei ein und weist sie Variablen zu.

Syntax

INPUT # *Dateinummer*, *Variablenliste*

Anmerkungen

Dateinummer ist die Nummer, unter der die Datei zur Eingabe geöffnet wurde. *Variablenliste* enthält die Namen der Variablen, denen die aus der Datei eingelesenen Werte zugewiesen werden. (Der Variablentyp muß mit dem Typ übereinstimmen, der durch den Variablennamen festgelegt ist.)

Anders als **INPUT** gibt **INPUT #** kein Fragezeichen aus.

Die Daten in der Datei müssen so aussehen, als wären sie als Antwort auf eine **INPUT**-Anweisung eingegeben worden. Bei numerischen Werten werden führende Leerzeichen, Wagenrückläufe und Zeilenvorschübe ignoriert. Das erste gelesene Zeichen, das weder ein Leerzeichen noch ein Wagenrücklauf oder Zeilenvorschub ist, wird als Anfangszeichen einer Zahl angenommen. Die Zahl endet mit einem Leerzeichen, Wagenrücklauf, Zeilenvorschub oder Komma.

Wenn BASIC die sequentielle Datei nach einer Zeichenkette durchsucht, ignoriert es ebenfalls führende Leerzeichen, Wagenrückläufe und Zeilenvorschübe. Wird bei der Eingabe eines numerischen Wertes oder einer Zeichenkette das Dateiende erreicht, so wird diese Eingabe unterdrückt.

Vergleichen Sie auch

INPUT, INPUT$

Beispiel

Das folgende Programm liest eine Serie von Testergebnissen aus einer sequentiellen Datei und berechnet den Ergebnisdurchschnitt:

```
DEFINT A-Z
OPEN "klass.dat" FOR INPUT AS #1
DO WHILE NOT EOF(1)
   Zaehl=Zaehl+1
   INPUT #1, Test
   Total=Total+Test
   PRINT Zaehl;Test
LOOP
PRINT
PRINT "Gesamtzahl Studenten: ";Zaehl;
PRINT "Ergebnisdurchschnitt:";Total/Zaehl
END
```

Ausgabe

```
1 97
2 84
3 63
4 89
5 100
Gesamtzahl Studenten: 5   Ergebnisdurchschnitt: 86.6
```

INPUT$-Funktion

Funktion

Gibt eine Folge von Zeichen an, die aus der angegebenen Datei gelesen werden.

Syntax

INPUT$ (*n*[,[#]*Dateinummer*])

Anmerkungen

n ist die Anzahl der aus der Datei zu lesenden Bytes (Zeichen). Die *Dateinummer* ist die zum Öffnen der Datei benutzte Nummer.

Wenn die Datei für Direktzugriff geöffnet ist, muß das Argument *n* kleiner gleich der Satzlänge sein, die durch die Klausel **LEN** in der Anweisung **OPEN** gesetzt ist (oder kleiner gleich 128, wenn die Satzlänge nicht gesetzt ist). Ist die angegebene Datei für binären oder sequentiellen Zugriff eröffnet, so muß *n* kleiner gleich 32.767 sein.

Falls *Dateinummer* nicht angegeben wird, werden die Zeichen aus dem Standard-Eingabegerät eingelesen. (Wenn die Eingabe nicht umgeleitet wurde, ist die Tastatur das Standard-Eingabegerät.)

Zur Neudefinierung der Standardeingabe oder -ausgabe für eine mit QuickBASIC erstellte ausführbare Datei können Sie die DOS-Umleitungssymbole (<, > oder >>) oder das Pipe-Symbol (|) verwenden. (Eine vollständige Beschreibung hierzu finden Sie im Handbuch zu Ihrem Betriebssystem.)

Es werden keine Zeichen auf den Bildschirm ausgegeben. Alle Kontrollzeichen, außer STRG+UNTBR, das die Ausführung der Funktion unterbricht, werden übergangen.

Beispiel

Das folgende Programm schreibt eine Datei auf den Bildschirm. Es benutzt **INPUT$**, um Zeichen für Zeichen zu lesen, wandelt das Zeichen, wenn nötig, um und zeigt es an.

```
'ASCII-Codes für TAB und Zeilenvorschub:
CONST HTAB = 9, LFEED = 10
INPUT "Welche Datei soll angezeigt werden";DateiName$
OPEN DateiName$ FOR INPUT AS #1
CLS
DO WHILE NOT EOF(1)
   ' Lies ein einzelnes Zeichen aus der Datei.
   S$=INPUT$(1,#1)
   ' Übertrage das Zeichen in eine Ganzzahl und lösche
   ' das höchste Bit, sodaß eine WordStar(R) Datei
   ' angezeigt werden kann.
   C=ASC(S$) AND &H7F
```

```
   ' Ist dies ein druckbares Zeichen?
   IF (C >= 32 AND C <= 126) OR C = HTAB OR C = LFEED THEN
      PRINT CHR$(C);
   END IF
LOOP
END
```

INSTR-Funktion

Funktion

Übergibt die Zeichenposition des ersten Vorkommens einer Zeichenkette in einer anderen Zeichenkette.

Syntax

INSTR ([*Start*,] *Zeichenkettenausdruck1*, *Zeichenkettenausdruck2*)

Anmerkung

Die folgende Liste beschreibt die Argumente der **INSTR**-Funktion:

Argument	***Beschreibung***
Start	Ein wahlweiser Offset, der die Position für den Beginn des Suchvorgangs setzt; *Start* muß im Bereich von 1 bis 32.767 liegen. Wenn *Start* nicht angegeben ist, beginnt die **INSTR**-Funktion die Suche am ersten Zeichen von *Zeichenkettenausdruck1*.
Zeichenkettenausdruck1	Die Zeichenkette, die durchsucht wird.
Zeichenkettenausdruck2	Die Zeichenkette, nach der zu suchen ist.

Die Argumente *Zeichenkettenausdruck1* und *Zeichenkettenausdruck2* können Zeichenkettenvariablen, Zeichenkettenausdrücke oder Zeichenkettenliterale sein.

Der von **INSTR** angegebene Wert hängt von folgenden Bedingungen ab:

Bedingung	***Angegebener Wert***
Zeichenkettenausdruck2 in *Zeichenkettenausdruck1* gefunden	Die Stelle, an der die Übereinstimmung gefunden wurde.
Start größer als Länge von *Zeichenkettenausdruck1*	0
Zeichenkettenausdruck1 ist Null-Zeichenkette	0
Zeichenkettenausdruck2 kann nicht gefunden werden	0
Zeichenkettenausdruck2 ist Null-Zeichenkette	*Start* (falls angegeben); andernfalls 1.

Die Länge von *Zeichenkettenausdruck1* können Sie mit der Funktion **LEN** ermitteln.

Beispiel

Das folgende Beispiel benutzt **INSTR** und **UCASE$** zur Suche nach Mr. und Mrs. oder Ms. in einem Namen, um auf das Geschlecht der Person zu schließen:

```
' Lies einen Namen.
DO
   INPUT "Geben Sie den Namen ein: ", Nm$
LOOP UNTIL LEN(Nm$)>=3
' Übertrage Klein- in Großbuchstaben:
Nm$ = UCASE$(Nm$)
' Suche nach MS, MRS oder MR, um Geschl$ zu setzen.
IF INSTR(Nm$,"MS") > 0 OR INSTR(Nm$,"MRS") > 0 THEN
    Geschl$ = "W"
ELSEIF INSTR(Nm$,"MR") > 0 THEN
    Geschl$ = "M"
ELSE
    ' Kann nicht auf Geschlecht schließen.
    ' Beim Benutzer nachfragen
    DO
        INPUT "Geschlecht eingeben (M/W): ", Geschl$
        Geschl$ = UCASE$(Geschl$)
    LOOP WHILE Geschl$ <> "M" AND Geschl$ <> "W"
END IF
```

Ausgabe

```
Geben Sie den Namen ein: Elisabeth Schlaumeier
Geschlecht eingeben (M/W): x
Geschlecht eingeben (M/W): W
```

INT-Funktion

Funktion

Gibt die größte ganze Zahl kleiner oder gleich *Numerischer Ausdruck* an.

Syntax

INT (*Numerischer Ausdruck*)

Anmerkungen

Die **INT**-Funktion entfernt die Nachkommastellen des Arguments.

Vergleichen Sie auch

CINT, FIX

Beispiel

Das folgende Beispiel vergleicht die Ausgabe der drei Funktionen, die numerische Daten in ganze Zahlen umwandeln:

```
PRINT "  N","INT(N)","CINT(N)","FIX(N)" : PRINT
FOR I% = 1 TO 6
    READ N
    PRINT N, INT(N), CINT(N), FIX(N)
NEXT
DATA 99.3, 99.5, 99.7, -99.3, -99.5, -99.7
```

Ausgabe

N	INT(N)	CINT(N)	FIX(N)
99.3	99	99	99
99.5	99	100	99
99.7	99	100	99
-99.3	-100	-99	-99
-99.5	-100	-100	-99
-99.7	-100	-100	-99

IOCTL-Anweisung

Funktion

Überträgt eine Steuerdaten-Zeichenkette zu einem Gerätetreiber.

Syntax

IOCTL [#]*Dateinummer, Zeichenkette*

Anmerkungen

Die *Dateinummer* ist die zum Öffnen des Gerätes verwendete BASIC-Dateinummer. Die *Zeichenkette* ist der Befehl, der zum Gerät geschickt wird. Befehle müssen dem Gerätetreiber entsprechende Befehle sein. In der Dokumentation des Gerätetreibers erfahren Sie, welche **IOCTL**-Befehle gültig sind. Eine **IOCTL**-Steuerdatenzeichenkette kann bis zu 32.767 Bytes lang sein.

Die **IOCTL**-Anweisung funktioniert nur, wenn die folgenden drei Bedingungen erfüllt sind.

1. Der Gerätetreiber ist installiert.
2. Der Gerätetreiber meldet, daß er **IOCTL**-Zeichenketten verarbeitet. Schlagen Sie in der Dokumentation des Treibers nach. Unter Verwendung des Interrupts &H21 mit der **CALL INTERRUPT**-Routine können Sie ebenfalls durch die DOS-Funktion &H44 auf Unterstützung von **IOCTL** testen. Weitere Informationen finden Sie in der Beschreibung zu **CALL INTERRUPT** und in *Microsoft MS-DOS Programmer's Reference*.
3. BASIC führt eine **OPEN**-Anweisung einer Datei auf diesem Gerät aus, die Datei bleibt geöffnet.

Die meisten DOS-Standard-Gerätetreiber verarbeiten keine **IOCTL**-Zeichenketten, und Sie müssen feststellen, ob ein spezieller Treiber diesen Befehl akzeptiert.

Vergleichen Sie auch

IOCTL$

Beispiel

Wenn Sie die Seitenlänge auf LPT1 auf 66 Zeilen pro Seite setzen wollen, kann Ihre Anweisung folgendermaßen aussehen:

```
OPEN "\GER\LPT1" FOR OUTPUT AS 1
IOCTL #1, "PL66"
```

IOCTL$-Funktion

Funktion

Empfängt eine Kontrolldaten-Zeichenkette von einem Gerätetreiber.

Syntax

IOCTL$ ([#]*Dateinummer*)

Anmerkungen

Die *Dateinummer* ist die BASIC-Dateinummer, die benutzt wurde, um das Gerät zu öffnen.

Die Funktion **IOCTL$** wird meist verwendet, um zu testen, ob eine **IOCTL**-Anweisung erfolgreich war oder nicht, oder um die aktuellen Statusinformationen abzufragen.

Mit **IOCTL$** können Sie ein Datenübertragungsgerät auffordern, die aktuelle Baud-Rate, Informationen über den letzten Fehler, die logische Zeilenbreite usw. zu ermitteln. Die genaue Information, die ausgegeben wird, ist abhängig vom entsprechenden Gerätetreiber.

Die Funktion **IOCTL$** funktioniert nur, wenn die folgenden drei Bedingungen erfüllt sind:

1. Der Gerätetreiber ist installiert.
2. Der Gerätetreiber meldet, daß er **IOCTL**-Zeichenketten verarbeitet. Schlagen Sie in der Dokumentation des Treibers nach. Unter Verwendung des Interrupts &H21 mit der **CALL INTERRUPT**-Routine können Sie ebenfalls durch die DOS-Funktion &H44 auf Unterstützung von **IOCTL** testen. Weitere Informationen finden Sie in der Beschreibung zu **CALL INTERRUPT**.
3. BASIC führt eine **OPEN**-Anweisung einer Datei zu diesem Gerät aus.

Hinweis BASIC-Geräte (**LPT1:**, **COM1:**, **COM2:**, **SCRN:**, **CONS:**) und DOS-Blockgeräte (**A:** bis **Z:**) unterstützen **IOCTL** nicht.

Vergleichen Sie auch

IOCTL

Beispiel

Das folgende Beispiel öffnet den Gerätetreiber "MASCHINE" und verwendet die **IOCTL$**-Funktion, um zu testen, ob der Ursprungsdatenmodus eingestellt ist.

```
OPEN "\GER\MASCHINE" FOR OUTPUT AS #1
IOCTL #1, "URSPR"   'Teilt dem Gerät mit, daß es sich
'um Ursprungsdaten handelt. Falls der Zeichentreiber
'"MASCHINE" aus dem Ursprungsdatenmodus in der
'Anweisung IOCTL "falsch" antwortet, wird die Datei
'geschlossen."
IF IOCTL$(1) = "0" THEN CLOSE 1
```

KEY-Anweisungen

Funktion

Weist den Funktionstasten Zeichenkettenwerte als Tastenbelegung zu, zeigt dann die Werte an und schaltet die Anzeigezeile für die Funktionstasten ein oder aus.

Syntax

KEY *n, Zeichenkettenausdruck*
KEY LIST
KEY ON
KEY OFF

Anmerkungen

Der Platzhalter *n* ist die Nummer, die die Funktionstaste darstellt. Die Werte für *n* sind 1 bis 10 für die Funktionstasten, und 30 und 31 für die Funktionstasten F11 und F12 auf der erweiterten 101-Tastatur. Der *Zeichenkettenausdruck* ist eine Zeichenkette von bis zu 15 Zeichen, die nach Betätigung der Funktionstaste ausgegeben wird. Wenn der *Zeichenkettenausdruck* länger als 15 Zeichen ist, werden die überzähligen Zeichen ignoriert.

Die Anweisung **KEY** erlaubt Ihnen eine benutzerdefinierte Funktionstastenbelegung - Zeichenketten, die ausgegeben werden, wenn Funktionstasten betätigt werden.

Die Zuweisung einer Null-Zeichenkette zu einer benutzerdefinierten Funktionstaste schaltet die Funktionstaste als benutzerdefinierte Funktionstaste aus.

Wenn die Funktionstastennummer außerhalb des Bereichs der zulässigen Nummern liegt, wird die Fehlermeldung `Unzulässiger Funktionsaufruf` angezeigt, und der vorhergehende Zeichenkettenausdruck für die Taste wird beibehalten.

Sie können die benutzerdefinierten Funktionstasten mit den Anweisungen **KEY ON**, **KEY OFF** und **KEY LIST** anzeigen:

Anweisung	*Beschreibung*
KEY ON	Zeigt die ersten sechs Zeichen des Zeichenkettenwertes der benutzerdefinierten Funktionstaste in der untersten Zeile des Bildschirms an.
KEY OFF	Löscht die Anzeige der benutzerdefinierten Funktionstasten in der untersten Zeile, damit diese vom Programm benutzt werden kann. Die Funktionstasten werden hierdurch nicht ausgeschaltet.
KEY LIST	Zeigt alle Werte der benutzerdefinierten Funktionstasten auf dem Bildschirm an, und zwar alle 15 Zeichen jeder Taste.

Wenn eine benutzerdefinierte Funktionstaste betätigt wird, ist die Auswirkung die gleiche, als hätte der Benutzer die Zeichenkette, die mit der benutzerdefinierten Funktionstaste zusammenhängt, eingegeben. **INPUT$**, **INPUT**, und **INKEY$** können alle verwendet werden, um die Zeichenkette, die durch Drücken der benutzerdefinierten Funktionstaste hervorgerufen wird, zu lesen.

Beispiel

Die folgenden Beispiele zeigen, wie eine Zeichenkette einer benutzerdefinierten Funktionstaste zugewiesen wird, und wie eine benutzerdefinierte Funktionstaste ausgeschaltet wird.

```
KEY 1, "MENÜ"+CHR$(13)     'Weist Funktionstaste 1 die
                           'Zeichenkette "MENÜ" zu,
                           'gefolgt von einem
                           'Wagenrücklauf

KEY 1, ""   'Schaltet benutzerdefinierte Funktionstaste
            '1 aus
```

Das folgende Programm benutzt die Anweisungen **KEY**, um ein Funktionstastenmenü zu erstellen. Durch das Betätigen von F1 wird beispielsweise dasselbe erreicht wie durch das Eingeben der Zeichenkette "Hin".

```
DIM KeyText$(3)
DATA Add, Loesche, Ende
' Weise F1 bis F3 benutzerdefinierte Zeichenketten zu.
FOR I=1 TO 3
   READ KeyText$(I)
   KEY I, KeyText$(I)+CHR$(13)
NEXT I
' Schreibe Menü.
PRINT "                     Hauptmenü" : PRINT
PRINT "     Zur Liste hinzufügen (F1)"
PRINT "     Aus der Liste löschen (F2)"
PRINT "     Ende (F3)"  :  PRINT
' Eingabe lesen und antworten.
DO
   LOCATE 7,1 : PRINT SPACE$(50);
   LOCATE 7,1 : INPUT "         Geben Sie Ihre _
                     Wahl ein:",R$
   SELECT CASE R$
      CASE "Hin", "Loesche"
         LOCATE 10,1 : PRINT SPACE$(15);
         LOCATE 10,1 : PRINT R$;
```

```
      CASE "Ende"
         EXIT DO
      CASE ELSE
         text$="Eingabe des ersten Wortes oder "
         LOCATE 10,1 : PRINT text$;
         PRINT  "Betätigen einer Taste."
   END SELECT
LOOP
```

KEY(*n*)-Anweisungen

Funktion

Startet und beendet die Verfolgung von angegebenen Tasten.

Syntax

KEY(*n*) ON
KEY(*n*) OFF
KEY(*n*) STOP

Anmerkungen

Das Argument *n* ist die Nummer einer Funktionstaste, einer Cursor-Richtungstaste oder einer vom Benutzer definierten Taste. (Informationen über die Zuweisung von Werten für die Belegung der Funktionstasten finden Sie unter der **KEY**-Anweisung.) Die Werte von *n* sind die folgenden:

Wert	***Taste***
1-10	Die Funktionstasten F1 - F10
11	NACH OBEN (↑)
12	NACH LINKS (←)
13	NACH RECHTS (→)
14	NACH UNTEN (↓)
15-25	Benutzerdefinierte Tasten
30-31	Die Funktionstasten F11 - F12 auf der erweiterten 101-Tastatur

NACH LINKS, NACH RECHTS, NACH OBEN und NACH UNTEN beziehen sich auf die Richtungstasten.

Sie können das Verfolgen von Tastenkombinationen aktivieren, indem Sie eine Variation der **KEY**-Anweisung benutzen.

KEY *n*, **CHR$** (*Tastaturkennzeichen*) + **CHR$** (*Abfragecode*)

Das Argument *n* liegt im Bereich von 15 - 25, um eine benutzerdefinierte Taste zu kennzeichnen.

Das *Tastaturkennzeichen* kann jede Kombination der folgenden Hexadezimalwerte sein:

Wert	*Taste*
&H00	Keine Tastaturkennzeichen
&H01-&H03	Beide UMSCHALTTASTEN
&H04	STRG-TASTE
&H08	ALT-TASTE
&H20	NUM-FESTSTELLTASTE
&H40	UMSCHALT-FESTSTELLTASTE (CAPS LOCK)
&H80	Zusätzliche Tasten der erweiterten 101-Tastatur

Sie können die Werte aufaddieren, um verschiedene Zustände der Umschalttasten zu prüfen. Zum Beispiel würde ein Wert von &H12 für *Tastaturkennzeichen* überprüfen, ob sowohl die STRG-TASTE als auch die ALT-TASTE betätigt werden.

Da Tastenverfolgung davon ausgeht, daß die linke und die rechte UMSCHALTTASTE identisch sind, können Sie entweder &H01, &H02 oder &H03 verwenden, um eine UMSCHALTTASTE anzuzeigen. Das Argument *Abfragecode* ist eine Zahl, die eine der 83 zu verfolgenden Tasten identifiziert, wie in der folgenden Tabelle gezeigt wird.

Beachten Sie, daß die an anderer Stelle beschriebene Anweisung **KEY** den Funktionstasten Zeichenketten oder Cursor-Richtungswerte zuweist und diese Werte anzeigt. Die Anweisungen **KEY ON** und **KEY OFF**, die diese Werte anzeigen und löschen, unterscheiden sich von den hier beschriebenen Ereignisverfolgungs-Anweisungen.

Die Anweisung **KEY**(*n*) **ON** aktiviert die Ereignisverfolgung für eine benutzerdefinierte Taste oder Cursor-Richtungstaste durch eine **ON KEY**-Anweisung. Wenn Sie während der aktivierten Erfassung in der Anweisung **ON KEY** eine Zeilennummer ungleich 0 angeben, überprüft BASIC, ob Sie **KEY**(*n*) betätigt haben. Wenn dies der Fall ist, führt BASIC das **GOSUB** in der Anweisung **ON KEY** aus. Der Text, der normalerweise mit einer Funktionstaste verknüpft ist, wird nicht ausgegeben.

Wenn Sie innerhalb der Umgebung arbeiten, prüft QuickBASIC zwischen Anweisungen auf Tastenbetätigungen. In selbständigen Programmen können Sie Prüfungen zwischen Zeilen festlegen. In Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen.

KEY(*n*) **OFF** deaktiviert die Ereignisverfolgung; auch wenn ein Ereignis eintritt, wird dieses nicht festgehalten.

KEY(*n*) **STOP** sperrt die Ereignisverfolgung, d. h., wenn Sie die angegebene Taste drücken, wird dies festgehalten und eine **ON KEY**-Ereignisverfolgung ausgeführt, sobald eine **KEY**(*n*) **ON**-Anweisung ausgeführt ist.

Taste	***Code in Hex***	***Taste***	***Code in Hex***	***Taste***	***Code in Hex***
ESC	01	STRG	1D	LEERTASTE	39
! oder 1	02	A	1E	UMSCHALT-FESTSTELLTASTE	3A
@ oder 2	03	S	1F	F1	3B
# oder 3	04	D	20	F2	3C
$ oder 4	05	F	21	F3	3D
% oder 5	06	G	22	F4	3E
^ oder 6	07	H	23	F5	3F
& oder 7	08	J	24	F6	40
* oder 8	09	K	25	F7	41
(oder 9	0A	L	26	F8	42
) oder 0	0B	: oder ;	27	F9	43
_ oder -	0C	" oder '	28	F10	44
+ oder =	0D	~ oder ‘	29	NUM-FESTSTELLTASTE	45
RÜCK-TASTE	0E	linke UMSCHALT-TASTE	2A	ROLLEN-FESTSTELLTASTE	46
TAB	0F	| oder \	2B	POS1 oder 7	47
Q	10	Z	2C	↑ oder 8	48
W	11	X	2D	BILD ↑ oder 9	49
E	12	C	2E	-	4A
R	13	V	2F	← oder 4	4B
T	14	B	30	5	4C
Y	15	N	31	→ oder 6	4D
U	16	M	32	+	4E

Taste	*Code in Hex*	*Taste*	*Code in Hex*	*Taste*	*Code in Hex*
I	17	< oder ,	33	ENDE oder 1	4F
O	18	> oder .	34	↓ oder 2	50
P	19	? oder /	35	BILD ↓ oder 3	51
{ oder [	1A	rechte UMSCHALT-TASTE	36	EINFG oder 0	52
} oder]	1B	DRUCK oder *	37	ENTF oder .	53
EINGABE-TASTE	1C	ALT	38		

Hinweis Die Abfragecodes in der Tabelle sind identisch mit der ersten Spalte der Tabelle der Abfragecodes in Anhang A, " ASCII-Zeichencodes und Tastaturabfragecodes". Die Codes in den anderen Spalten der Tabelle im Anhang sollten nicht für die Tastenverfolgung verwendet werden.

Vergleichen Sie auch

ON *Ereignis*

Beispiel

Dieses Beispiel verfolgt die NACH UNTEN-Richtungstaste und STRG+S (STRG-TASTE und kleines "s"). Zur Verfolgung der Kombination der STRG-TASTE und des großgeschriebenen "s" müssen Sie STRG+UMSCHALT+S und STRG+NUM-FESTSTELLTASTE+S verfolgen.

```
I = 0
PRINT "Drücken Sie die Richtungstaste NACH UNTEN zum";
PRINT "Beenden."
KEY 15,CHR$(&H04)+CHR$(&H1f)
KEY(15) ON        'Taste STRG+s wird jetzt erfaßt
KEY(14) ON        'Taste "NACH UNTEN" ( » )wird jetzt erfaßt
ON KEY(15) GOSUB TastErf
ON KEY(14) GOSUB EndProg
Endlos: GOTO Endlos     'Endlos-Schleife
```

```
TastErf:  'Zählt, wie oft die Taste STRG+s gedrückt
          'wird
      I = I + 1
RETURN

EndProg:
   PRINT "STRG+s wurde "; I; "mal gedrückt"
   END
RETURN
```

KILL-Anweisung

Funktion

Löscht eine Datei von der Diskette/Platte.

Syntax

KILL *Dateiangabe*

Anmerkungen

Die **KILL**-Anweisung ist ähnlich den DOS-Befehlen **ERASE** oder **DEL**.

KILL wird für alle Disketten-/Plattendateitypen benutzt: Programmdateien, Direktzugriffsdateien und sequentielle Dateien. *Dateiangabe* kann das Fragezeichen (?) oder den Stern (*) als Joker-Zeichen enthalten. Ein Fragezeichen entspricht jedem einzelnen Zeichen im Dateinamen oder seiner Erweiterung. Ein Stern entspricht einem oder mehreren Zeichen ab seiner Position.

Sie können **KILL** nur zum Löschen von Dateien verwenden. Zum Löschen von Verzeichnissen dient der Befehl **RMDIR**.

Die Verwendung von **KILL** zum Löschen einer momentan geöffneten Datei führt zu der Fehlermeldung `Datei bereits geöffnet`.

Warnung Seien Sie äußerst vorsichtig, wenn Sie Joker-Zeichen in Verbindung mit **KILL** benutzen. Bei der Verwendung der Joker-Zeichen können Sie unbeabsichtigt Dateien löschen.

Beispiele

Die folgenden Beispiele zeigen die Wirkung von Joker-Zeichen in Verbindung mit **KILL**:

```
KILL "DATA1?.DAT"  'Löscht alle Dateien, deren
                   'Stammname sechs Zeichen lang ist,
                   'mit DATA1 anfängt und die
                   'Erweiterung .DAT hat
KILL "DATA1.*"     'Löscht alle Dateien mit dem
                   'Stammnamen DATA1 und einer
                   'beliebigen Erweiterung
KILL "\GREG\*.DAT" 'Löscht alle Dateien mit der
                   'Erweiterung .DAT im Unterverzeichnis
                   'GREG
```

Das folgende Programm löscht die in der Befehlszeile angegebene Datei:

```
DEFINT A-Z
ON ERROR GOTO Fehlerbehandl 'Einschalten
                            'Fehlerbehandlung.
DateiName$ = COMMAND$       'Lies den Dateinamen.
KILL DateiName$
END
Fehlerbehandl:
   Zahl = ERR
   IF Zahl = 53 THEN
      PRINT "Konnte ";DateiName$;" nicht löschen";
      PRINT "; Datei existiert nicht im aktuellen ";
      PRINT "Verzeichnis"
   ELSE
      PRINT "Nicht wiederherstellbarer Fehler:";Zahl
      ON ERROR GOTO 0    'ON ERROR GOTO Null bricht das
                         'Programm ab.
   END IF
RESUME NEXT
```

LBOUND-Funktion

Funktion

Ermittelt die untere Grenze (den kleinsten vorhandenen Index) für die angegebene Dimension eines Datenfeldes.

Syntax

LBOUND (*Datenfeld* [, *Dimension*])

Anmerkungen

Die **LBOUND**-Funktion wird mit der Funktion **UBOUND** verwendet, um die Größe eines Datenfeldes zu ermitteln. **LBOUND** hat folgende Argumente:

Argument	***Beschreibung***
Datenfeld	Der Name des dimensionierten Datenfeldes.
Dimension	Eine ganze Zahl von 1 bis zur Anzahl der Dimensionen im *Datenfeld*, die angibt, von welcher Dimension die untere Grenze ausgegeben wird.

LBOUND gibt folgende Werte für ein wie folgt dimensioniertes Datenfeld aus:

```
DIM A(1 TO 100, 0 TO 50, -3 TO 4)
```

Aufruf	***Zurückgegebener Wert***
`LBOUND(A,1)`	`1`
`LBOUND(A,2)`	`0`
`LBOUND(A,3)`	`-3`

Die vorgegebene untere Grenze für jede Dimension ist, je nach Festlegung der Anweisung **OPTION BASE**, entweder 0 oder 1. Ist **OPTION BASE** 0, so ist die vorgegebene untere Grenze 0, mit **OPTION BASE** 1 ist die vorgegebene untere Grenze 1.

Datenfelder, die unter Verwendung der **TO**-Klausel in der **DIM**-Anweisung dimensioniert sind, können jeden ganzzahligen Wert als untere Grenze haben.

Für eindimensionale Datenfelder können Sie die verkürzte Syntax **LBOUND**(*Datenfeld*) verwenden, weil der Standardwert für *Dimension* 1 ist.

Verwenden Sie die Funktion **UBOUND** zur Ermittlung der oberen Grenze einer Datenfelddimension.

Vergleichen Sie auch

UBOUND

Beispiel

Die Funktionen **LBOUND** und **UBOUND** können verwendet werden, um die Größe eines an ein Unterprogramm übergebenen Datenfeldes zu bestimmen, wie im folgenden Programmausschnitt gezeigt wird:

```
CALL Drckmat(Array())
.
.
.
SUB Drckmat(A(2)) STATIC
   ' Äußere Schleife überprüft Zeile (Dimension 1).
   FOR I% = LBOUND(A,1) TO UBOUND(A,1)
      ' Innere Schleife überprüft Spalte
      ' (Dimension 2).
      FOR J% = LBOUND (A,2) TO UBOUND(A,2)
         PRINT A(I%,J%);" ";
      NEXT J%
      PRINT:PRINT
   NEXT I%
END SUB
```

LCASE$-Funktion

Funktion

Gibt einen Zeichenkettenausdruck mit sämtlichen Zeichen als Kleinbuchstaben an.

Syntax

LCASE$ (*Zeichenkettenausdruck*)

Anmerkungen

Die **LCASE$**-Funktion faßt eine Zeichenkettenvariable, eine Zeichenkettenkonstante oder einen Zeichenkettenausdruck als ihr einziges Argument auf.

LCASE$ arbeitet sowohl mit Zeichenketten variabler als auch fester Länge.

LCASE$** und **UCASE$ sind hilfreich bei Zeichenketten-Vergleichsoperationen, bei denen die Überprüfung unabhängig von der Groß-/Kleinschreibung sein soll.

Vergleichen Sie auch

UCASE$

Beispiel

Das folgende Beispiel wandelt Großbuchstaben in einer Zeichenkette in Kleinbuchstaben um:

```
' Programm zum Übertragen in Kleinbuchstaben
  READ Wort$
  PRINT LCASE$(Wort$);
  DATA "EINE ZEICHENKETTE in Kleinbuchstaben."
```

Ausgabe

```
eine zeichenkette in kleinbuchstaben.
```

LEFT$-Funktion

Funktion

Gibt eine Zeichenkette an, welche die *n* am weitesten links stehenden Zeichen einer Zeichenkette beinhaltet.

Syntax

LEFT$ (*Zeichenkettenausdruck,n*)

Anmerkungen

Das Argument *Zeichenkettenausdruck* kann jede Zeichenkettenvariable, Zeichenkettenkonstante oder jeder Zeichenkettenausdruck sein.

Das Argument *n* ist ein numerischer Ausdruck im Bereich 0 bis 32.767, der anzeigt, wieviele Zeichen auszugeben sind.

Wenn *n* größer als die Anzahl der Zeichen in *Zeichenkettenausdruck* ist, wird die gesamte Zeichenkette angegeben. Zur Ermittlung der Anzahl der Zeichen in *Zeichenkettenausdruck* dient die Funktion **LEN** (*Zeichenkettenausdruck*).

Ist *n* gleich Null, wird die Null-Zeichenkette (Länge Null) angegeben.

Vergleichen Sie auch

MID$, RIGHT$

Beispiel

Dieses Beispiel gibt die fünf am weitesten links stehenden Zeichen von A$ aus:

```
A$="BASIC-HANDBUCH"
B$=LEFT$(A$,5)
PRINT B$
```

Ausgabe

```
BASIC
```

LEN-Funktion

Funktion

Gibt die Anzahl der Zeichen in einer Zeichenkette oder die Anzahl der von einer Variablen benötigten Bytes an.

Syntax

LEN (*Zeichenkettenausdruck*)

LEN (*Variable*)

Anmerkungen

In der ersten Form gibt **LEN** die Anzahl der Zeichen in dem Argument *Zeichenkettenausdruck* an.

Die zweite Syntax gibt die Anzahl der Bytes an, die die BASIC-Variable benötigt. Diese Syntax ist besonders nützlich zur Festlegung der Größe des Puffers einer Direktzugriffsdatei.

Beispiel

Das folgende Beispiel druckt die Größen von BASIC-Variablen unterschiedlicher Typen und die Länge einer Zeichenkette aus:

```
TYPE EmpRec
   EmpName AS STRING*20
   EmpNum  AS INTEGER
END TYPE
DIM A AS INTEGER, B AS LONG, C AS SINGLE, D AS DOUBLE
DIM E AS EmpRec
PRINT "Integer:" LEN(A)
PRINT "Long:" LEN(B)
PRINT "Single:" LEN(C)
PRINT "Double:" LEN(D)
PRINT "EmpRec:" LEN(E)
PRINT "Eine Zeichenkette:" LEN("Eine Zeichenkette.")
END
```

Ausgabe

```
Integer: 2
Long: 4
Single: 4
Double: 8
EmpRec 22
Eine Zeichenkette: 18
```

LET-Anweisung

Funktion

Weist einer Variablen den Wert eines Ausdrucks zu.

Syntax

[**LET**] *Variable* = *Ausdruck*

Anmerkungen

Beachten Sie, daß das Wort **LET** wahlfrei ist. Das Gleichheitszeichen in einer Anweisung genügt, um QuickBASIC mitzuteilen, daß es sich um eine Zuweisung handelt.

Die **LET**-Anweisung kann mit Verbundvariablen nur benutzt werden, wenn beide Variablen vom selben benutzerdefinierten Typ sind. Benutzen Sie die **LSET**-Anweisung, um Verbundvariablen unterschiedlich benutzerdefinierter Typen zuzuweisen.

Vergleichen Sie auch

LSET

Beispiel

In diesen beiden Beispielen führen die entsprechenden Zeilen dieselbe Funktion aus:

```
LET D=12
LET E=12-2
LET F=12-4
LET SUMME=D+E+F
.
.
.
```

oder

```
D=12
E=12-2
F=12-4
SUMME=D+E+F
.
.
.
```

LINE-Anweisung

Funktion

Zeichnet eine Gerade oder ein Rechteck auf den Bildschirm.

Syntax

LINE [[**STEP**](*x1,y1*)]-[**STEP**](*x2,y2*)[,[*Farbe*][,[**B**[**F**]][,*Struktur*]]]

Anmerkungen

Die Koordinaten (*x1,y1*) und (*x2,y2*) geben die Endpunkte der Geraden an; ihre Reihenfolge ist unwichtig, da eine Gerade von (10,20) bis (120,130) dasselbe ist wie eine Gerade von (120,130) bis (10,20).

Die Option **STEP** gibt die spezifizierten Koordinaten relativ zu dem allerletzten Punkt an, im Gegensatz zu absolut gezeichneten Koordinaten. Wenn beispielsweise der letzte Punkt, auf den das Programm sich bezieht, (10,10) ist, dann zeichnet

```
LINE  -STEP (10,5)
```

eine Gerade von (10,10) zu dem Punkt mit der x-Koordinate 10 + 10 und der y-Koordinate 10 + 5 oder (20,15).

Eine andere Möglichkeit, einen neuen letzten Punkt einzuführen, ist die Initialisierung des Bildschirms mit den Anweisungen **CLS** und **SCREEN**. Die Verwendung der Anweisungen **PSET, PRESET, CIRCLE** und **DRAW** führt ebenfalls einen neuen letzten Punkt ein.

Nachfolgend werden Variationen des **STEP**-Arguments gezeigt. Für die folgenden Beispiele wird als zuletzt gezeichneter Punkt (10,10) angenommen:

Anweisung	*Beschreibung*
`LINE -(50,50)`	Zeichnet von (10,10) bis (50,50).
`LINE -STEP(50,50)`	Zeichnet von (10,10) bis (60,60), das bedeutet, bis 10 plus Offset 50.
`LINE (25,25) -STEP(50,50)`	Zeichnet von (25,25) bis (75,75); das bedeutet, von 25 plus Offset 50.
`LINE STEP(25,25) -STEP(50,50)`	Zeichnet von (35,35) bis (85,85), das bedeutet, von 10 plus Offset 25 zu dem Punkt plus Offset 50.
`LINE STEP (25,25) - (50,50)`	Zeichnet von (35,35) bis (50,50), das bedeutet, von 10 plus Offset 25 bis absolut 50.

Farbe ist die Nummer der Farbe, in der die Gerade gezeichnet wird. (Bei Verwendung der Optionen **B** oder **BF** wird das Rechteck in dieser Farbe gezeichnet.) Informationen über zulässige Farben finden Sie unter der Anweisung **SCREEN**.

Die Option **B** zeichnet ein Rechteck mit den Punkten (*x1*,*y1*) und (*x2*,*y2*), welche die diagonal gegenüberliegenden Ecken angeben.

Die Option **BF** zeichnet ein ausgefülltes Rechteck. Diese Option ist ähnlich zu der Option **B**. **BF** füllt außerdem das Innere des Rechtecks mit der gewählten Farbe aus.

Die *Struktur* ist eine ganzzahlige 16-Bit-Maske, mit der Punkte auf dem Bildschirm dargestellt werden. Die Angabe des *Struktur*-Arguments wird als "Geradenstruktur" bezeichnet. Bei der Geradenstruktur liest **LINE** die Bits in *Struktur* von links nach rechts. Wenn ein Bit 0 ist, wird kein Punkt dargestellt; ist das Bit 1, so wird ein Punkt dargestellt. Nach der Darstellung eines Punktes wählt **LINE** die nächste Bitposition in *Struktur*.

Da ein 0-Bit in *Struktur* keine Änderung am Punkt auf dem Bildschirm bewirkt, wünschen Sie vielleicht, eine Hintergrundgerade vor einer mit Struktur gezeichneten Gerade zu zeichnen, um einen bekannten Hintergrund zu erhalten. Struktur wird für normale Geraden und Rechtecke benutzt, hat aber keine Wirkung auf ausgefüllte Rechtecke.

Wenn die Koordinaten einen Punkt angeben, der sich nicht im aktuellen Darstellungsfeld befindet, wird das Geradensegment auf das Darstellungsfeld begrenzt.

Vergleichen Sie auch

SCREEN; Kapitel 5, "Grafiken", in *Programmieren in Basic: Ausgewählte Themen.*

Beispiele

Die nachstehenden Beispiele sind verschiedene **LINE**-Anweisungen, die einen Bildschirm von 320 x 200 Bildpunkten (Breite x Höhe) voraussetzen:

```
SCREEN 1               ' Stellt den Bildschirmmodus ein
LINE-(X2,Y2)           ' Zeichnet eine Gerade (in der
                       ' Textfarbe) vom letzten Punkt zu
                       ' X2,Y2
LINE(0,0)-(319,199)      ' Zeichnet eine diagonale
                         ' Gerade über den
                         ' Bildschirm (abwärts)
LINE(0,100)-(319,100)    ' Zeichnet eine horizontale
                         ' Gerade über den Bildschirm
LINE(10,10)-(20,20),2    'Zeichnet eine Gerade in
                         'Farbe 2
FOR X=0 to 319
    LINE(X,0)-(X,199),X AND 1
NEXT                       ' Zeichnet eine gestrichelte
                           ' Gerade auf einem
                           ' monochromen Bildschirm
LINE (0,0)-(100,100),,B    'Zeichnet ein Rechteck
                           '(beachten Sie,
                           'daß die Farbe nicht
                           'angegeben ist)
LINE STEP(0,0)-STEP(200,200),2,BF  'Malt ein Rechteck
                                   'in der Farbe 2 aus
                                   '(die Koordinaten
                                   'werden als Offsets
                                   'in der STEP-Option
                                   'angegeben)
LINE(0,0)-(160,100),3,,&HFF00  'Zeichnet eine
                               'unterbrochene Gerade
                               'von der linken oberen
                               'Ecke zur Mitte des
                               'Bildschirms in Farbe 3
```

LINE INPUT-Anweisung

Funktion

Gibt eine ganze Zeile (bis zu 255 Zeichen) in eine Zeichenkettenvariable ein, ohne dabei Begrenzer zu verwenden.

Syntax

LINE INPUT [;]["*Anfrage*";] *Zeichenkettenvariable*

Anmerkungen

Anfrage ist eine Zeichenkettenkonstante, die auf dem Bildschirm angezeigt wird, bevor die Eingabe angenommen wird. Ein Fragezeichen wird nur ausgegeben, wenn es Teil von *Anfrage* ist. Alle Eingaben ab dem Ende von *Anfrage* bis zum Wagenrücklauf werden der *Zeichenkettenvariable* zugewiesen.

Ein Semikolon direkt nach **LINE INPUT** hält den Cursor solange auf der gleichen Zeile, bis der Benutzer die EINGABETASTE betätigt.

LINE INPUT benutzt dieselben Editierzeichen wie **INPUT**.

Vergleichen Sie auch

INPUT

Beispiel

Das folgende Programm erlaubt dem Benutzer die Eingabe von Text in eine Notizdatei. Mit **LINE INPUT** können Sie alle Zeichen, einschließlich der Begrenzungszeichen (wie z. B. ein Komma), eingeben, die in einer normalen **INPUT**-Anweisung Begrenzer sind.

```
'Öffnet und schreibt solange Zeilen in eine
'Notizdatei, bis Sie eine Leerzeile eingeben
DO
   CLS
   PRINT "Geben Sie Text ein. Zum Beenden drücken";
   PRINT " Sie die <EINGABETASTE> ohne neuen Text
           einzugeben."
   PRINT
```

```
    OPEN "NOTIZ.TXT" FOR OUTPUT AS #1
    ' Nimmt Zeilen solange an, bis eine Leerzeile
    ' eingegeben wird.
    DO
       LINE INPUT "->";InLine$
       IF InLine$ <> "" THEN PRINT #1, InLine$
    LOOP WHILE InLine$ <> ""
    CLS : CLOSE #1
    ' Drucke Notizen aus und prüfe, ob sie korrekt
    ' sind.
    OPEN "NOTIZ.TXT" FOR INPUT AS #1
    PRINT "Ihre Eingabe ist: " : PRINT
    DO WHILE NOT EOF(1)
        LINE INPUT #1, InLine$
        PRINT InLine$
     LOOP
     CLOSE #1
     PRINT : INPUT "Ist dies richtig (J/N)"; R$
LOOP WHILE UCASE$(R$)="N"
END
```

LINE INPUT #-Anweisung

Funktion

Liest eine ganze Zeile ohne Begrenzer aus einer sequentiellen Datei in eine Zeichenkettenvariable ein.

Syntax

LINE INPUT # *Dateinummer*, *Zeichenkettenvariable*

Anmerkungen

Dateinummer ist die zum Öffnen der Datei benutzte Nummer. *Zeichenkettenvariable* ist die Variable, der die Zeile zugewiesen wird.

Die **LINE INPUT #**-Anweisung liest alle Zeichen in der sequentiellen Datei bis zu einem Wagenrücklauf ein. Anschließend überspringt sie die Folge Wagenrücklauf-Zeilenvorschub. Die nächste **LINE INPUT #**-Anweisung liest alle Zeichen bis zum nächsten Wagenrücklauf ein.

LINE INPUT # ist besonders nützlich, wenn jede Zeile einer Datei in Felder aufgeteilt wurde, oder wenn eine Textdatei Zeile für Zeile gelesen wird.

Vergleichen Sie auch

INPUT$, LINE INPUT

Beispiel

Das folgende Programm verwendet **LINE INPUT #**, um in eine Datei eingegebene Daten zurückzuschreiben.

```
OPEN "LISTE" FOR OUTPUT AS #1
PRINT "KUNDENINFORMATION:"
' Lies Kundeninformation.
DO
   PRINT
   INPUT "   NACHNAME:    ",NName$
   INPUT "   VORNAME:     ",VName$
   INPUT "   ALTER:       ",Alter$
   INPUT "   GESCHLECHT: ",Geschl$
   Geschl$=UCASE$(Geschl$)
   WRITE #1, NName$, VName$, Alter$, Geschl$
   INPUT "Weiteren Satz hinzufügen?";R$
LOOP WHILE UCASE$(R$)="J"
CLOSE #1
' Schreib die Datei zurück.
OPEN "LISTE" FOR INPUT AS #1
CLS
PRINT "Sätze in der Datei:" : PRINT
DO WHILE NOT EOF(1)
   LINE INPUT #1, Rec$ 'Liest Sätze aus Datei mit
                       'LINE INPUT #. Gibt die Sätze am
                       'Bildschirm aus.
   PRINT Rec$
LOOP
```

Eingabe

```
KUNDENINFORMATION:
   NACHNAME:    Krämer
   VORNAME:     Petra
   ALTER:       28
   Geschlecht: w
Weiteren Satz hinzufügen? J
   NACHNAME:    Walter
   VORNAME:     Peter
   ALTER:       28
   Geschlecht: m
Weiteren Satz hinzufügen? N
```

Ausgabe

```
Sätze in der Datei:
"Krämer",Petra","28","W"
"Walter",Peter","28","M"
```

LOC-Funktion

Funktion

Gibt die aktuelle Position in der Datei an.

Syntax

LOC (*Dateinummer*)

Anmerkungen

Dateinummer ist die in der **OPEN**-Anweisung zum Öffnen der Datei benutzte Nummer.

Bei Direktzugriffsdateien gibt die **LOC**-Funktion die Nummer des letzten Satzes an, der aus der Datei gelesen oder in die Datei geschrieben wurde.

Bei sequentiellen Dateien, gibt **LOC** die aktuelle Byte-Position in der Datei, dividiert durch 128, an.

Bei Dateien im Binär-Modus gibt **LOC** die Position von dem letzten geschriebenen oder gelesenen Byte an.

Für ein **COM**-Gerät gibt **LOC**(*Dateinummer*) die Anzahl der in der Eingabe-Warteschlange stehenden Zeichen an. Der Wert, der ausgegeben wird, hängt davon ab, ob das Gerät im ASCII- oder im Binär-Modus geöffnet wurde. Im ASCII-Modus stoppen maschinenorientierte Routinen das Anhängen von Zeichen, sobald das Dateiende-Zeichen empfangen wurde. Das Dateiende selbst wird nicht an die Warteschlange angehängt und kann nicht eingelesen werden. Ein Versuch zum Lesen des Dateiendes führt zu der Fehlermeldung `Eingabe nach logischem Ende`. Im Binär-Modus wird das Dateiende-Zeichen ignoriert und die gesamte Datei kann eingelesen werden.

Die **LOC**-Funktion kann nicht bei **SCRN:**-, **KYBD:**- oder **LPT***n***:**-Geräten benutzt werden.

Vergleichen Sie auch

OPEN COM

Beispiel

Die folgende Zeile stoppt das Programm, wenn die aktuelle Dateiposition über 50 ist.

```
IF LOC(1) > 50 THEN STOP
```

LOCATE-Anweisung

Funktion

Bewegt den Cursor zur angegebenen Position.

Syntax

LOCATE [*Zeile*][,[*Spalte*][,[*Cursor*][,[*Anfang*][,*Ende*]]]]

Anmerkungen

Die folgende Liste beschreibt die Argumente der **LOCATE**-Anweisung:

Argument	*Beschreibung*
Zeile	Die Nummer der Zeile auf dem Bildschirm; *Zeile* ist ein numerischer Ausdruck, der eine Ganzzahl angibt. Wenn *Zeile* nicht festgelegt wird, ändert sich die Zeile nicht.
Spalte	Die Nummer der Spalte auf dem Bildschirm; *Spalte* ist ein numerischer Ausdruck, der eine Ganzzahl angibt. Wenn *Spalte* nicht angegeben wird, ändert sich die Spaltenposition nicht.
Cursor	Ein Boolescher Wert, der angibt, ob der Cursor sichtbar oder unsichtbar sein soll. Wert 0 (Null) bedeutet Cursor *aus* und Wert 1 Cursor *ein*.
Anfang	Die Anfangszeile des Cursors auf dem Bildschirm. Es muß ein numerischer Ausdruck sein, der eine Ganzzahl angibt.
Ende	Die Endezeile des Cursors auf dem Bildschirm. Es muß ein numerischer Ausdruck sein, der eine Ganzzahl angibt.

Sie können jegliche Argumente der Anweisung auslassen. Wenn Sie *Zeile* oder *Spalte* auslassen, läßt **LOCATE** den Cursor in der Zeile und Spalte, an der er durch eine vorhergehende **LOCATE**- oder eine vorhergehende Eingabe- oder Ausgabe-Anweisung bewegt wurde, je nachdem welche zuletzt eingetreten ist. Wenn Sie andere Argumente auslassen, nimmt BASIC den vorhergehenden Wert des Arguments an.

Beachten Sie, daß die *Anfangs*- und *Ende*-Zeile die Bildröhren-Abtastlinien sind, die festlegen, welche Punkte auf dem Bildschirm hell dargestellt werden. Ein großer Bereich zwischen Anfangs- und Endezeile erzeugt einen größeren Cursor, z. B. einen solchen, der einen ganzen Zeichenblock einnimmt. Wenn *Anfang* kleiner als *Ende* ist, erstellt **LOCATE** einen zweiteiligen Cursor.

Wird die *Anfangs*-Zeile angegeben, die *Ende*-Zeile jedoch weggelassen, so nimmt *Ende* denselben Wert wie *Anfang* an.

Die letzte Bildschirmzeile ist für die Anzeige der benutzerdefinierten Funktionstasten reserviert und für den Cursor nur zugänglich, wenn die Anzeige ausgeschaltet ist (**KEY OFF**) und **LOCATE** in Verbindung mit **PRINT** zum Schreiben in dieser Zeile benutzt wird.

Vergleichen Sie auch

CSRLIN, POS

Beispiele

Die folgenden Anweisungen zeigen die Auswirkungen verschiedener **LOCATE**-Anweisungen auf den Cursor:

```
LOCATE 1,1      'Bewegt den Cursor zur oberen linken
                'Ecke des Bildschirms
LOCATE,,1       'Macht den Cursor sichtbar; Position
                'bleibt unverändert
LOCATE,,,7      'Position und Sichtbarkeit des Cursors
                'bleiben unverändert; setzt den Cursor
                'an den unteren Rand des Zeichenfeldes,
                'beginnend und endend mit Abtastzeile 7
LOCATE 5,1,1,0,7      'Bewegt den Cursor in Zeile 5,
                      'Spalte 1; macht den Cursor
                      'sichtbar; der Cursor nimmt den
                      'gesamten Zeichenraum, beginnend
                      'mit Abtastzeile 0 und endend mit
                      'Abtastzeile 7, ein
```

Das nachstehende Beispiel gibt ein Menü auf dem Bildschirm aus und wartet anschließend auf eine Eingabe im zulässigen Bereich (1 - 4). Wenn eine Zahl außerhalb dieses Bereichs eingegeben wird, wiederholt das Programm die Eingabeaufforderung.

```
CONST FALSCH=0, WAHR=NOT FALSCH
DO
    CLS
    PRINT "HAUPTMENÜ" : PRINT
    PRINT "1) Sätze hinzufügen"
    PRINT "2) Anzeigen/Aktualisieren/Löschen ";
    PRINT "eines Satzes"
    PRINT "3) Schreibe Liste von Leuten,";
    PRINT "die im Hotel sind"
    PRINT "4) Beende Programm"
    ' Verändere Cursor zu einem Block.
    LOCATE ,,1,1,12
    LOCATE 12,1
    PRINT "Wie heißt Ihre Wahl?";
```

```
    DO
       CH$ = INPUT$(1)
    LOOP WHILE (CH$ < "1" OR CH$ > "4")
    PRINT CH$
    ' Rufe das entsprechende Unterprogramm auf.
    SELECT CASE VAL(CH$)
       CASE 1
          CALL Hinzufuegen
       CASE 2
          CALL Suchen
       CASE 3
          CALL Hotel
       CASE 4
          CALL Verlassen
    END SELECT
LOOP WHILE NOT ENDPROG
.
.
.
END
```

LOCK...UNLOCK-Anweisungen

Funktion

Steuert den Zugriff von anderen Prozessen auf eine gesamte oder einen Teil einer geöffneten Datei.

Syntax

LOCK [#]*Dateinummer*[,{*Satz* | [*Anfang*] **TO** *Ende*}]

.

.

.

UNLOCK [#]*Dateinummer*[,{*Satz* | [*Anfang*] **TO** *Ende*}]

Anmerkungen

Diese Anweisungen werden in Netzwerkumgebungen verwendet, in denen mehrere Prozesse Zugriff auf dieselbe Datei erfordern können. Die Anweisungen **LOCK** und **UNLOCK** haben folgende Argumente:

Argument	***Beschreibung***
Dateinummer	Die Nummer, unter der die Datei geöffnet wurde.
Satz	Die Nummer des zu sperrenden Datensatzes oder Bytes; *Satz* kann jede Zahl zwischen 1 und 2.147.483.647 (2^{31}-1) sein. Ein Satz kann bis zu 32.767 Bytes lang sein.
Anfang	Die Nummer des ersten zu sperrenden Satzes oder Bytes.
Ende	Die Nummer des letzten zu sperrenden Satzes oder Bytes.

Für Dateien im Binär-Modus repräsentieren die Argumente *Satz*, *Anfang* und *Ende* die Anzahl der Bytes relativ zum Anfang der Datei. Das erste Byte in einer Datei ist Byte 1.

Für Direktzugriffsdateien repräsentieren die Argumente *Satz*, *Anfang* und *Ende* die Anzahl der Sätze relativ zum Anfang der Datei. Der erste Satz in einer Datei ist Satz 1.

Die Anweisungen **LOCK** und **UNLOCK** werden stets als Paare benutzt. Die Argumente von **LOCK** und **UNLOCK** müssen exakt miteinander übereinstimmen, wenn Sie sie verwenden. Siehe zweites Beispiel unten.

Wenn Sie nur einen Satz angeben, wird nur dieser gesperrt oder freigegeben. Geben Sie einen Satzbereich an und lassen den Anfangssatz (*Anfang*) weg, so werden alle Sätze ab dem ersten Satz bis zum Ende des Bereichs (*Ende*) gesperrt oder freigegeben. **LOCK** ohne Satzargument sperrt die gesamte Datei, während **UNLOCK** ohne Satzargument die gesamte Datei freigibt.

Wurde die Datei für sequentielle Ein- oder Ausgabe eröffnet, so wirken sich **LOCK** und **UNLOCK** auf die gesamte Datei aus, ungeachtet des durch *Anfang* und durch *Ende* festgelegten Bereichs. **LOCK** und **UNLOCK** funktionieren nur während der Ausführung, wenn Sie DOS-Versionen verwenden, die das Arbeiten im Netzwerk unterstützen (Version 3.1 oder später). Außerdem muß jedes Terminal (oder das Installationsprogramm für das Netzwerk) das DOS-Programm **SHARE.EXE** ausführen, um eine Sperroperation zu ermöglichen. Frühere DOS-Versionen erzeugen eine Fehlermeldung `Erweiterte Eigenschaft nicht verfügbar`, wenn **LOCK** oder **UNLOCK** ausgeführt werden.

Wichtig Vergessen Sie nicht, mit einer **UNLOCK**-Anweisung alle Sperren aufzuheben, bevor Sie eine Datei schließen oder Ihr Programm beenden. Fehlende Aufhebungen von Sperren führen zu unvorhersehbaren Ergebnissen.

Die Argumente von **LOCK** und **UNLOCK** müssen genau übereinstimmen.

Wenn Sie versuchen, auf eine gesperrte Datei zuzugreifen, können folgende Fehlermeldungen erscheinen:

```
Falsche Datensatznummer
Zugriff nicht gestattet
```

Beispiele

Diese Beispiele gehen von einer Direktzugriffsdatei aus.

Die nachstehende Anweisung sperrt die gesamte als Nummer 2 eröffnete Datei:

```
LOCK #2
```

Die nachstehende Anweisung sperrt nur Satz 32 in Datei Nummer 2:

```
LOCK #2, 32
```

Die folgende Anweisung sperrt die Sätze 1 - 32 in Datei Nummer 2:

```
LOCK #2, TO 32
```

Die beiden folgenden **UNLOCK**-Anweisungen heben die Sperren der Sätze, die von den vorhergehenden **LOCK**-Anweisungen gesperrt wurden, auf:

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 4
UNLOCK #1, 5 TO 8
```

Die folgende **UNLOCK**-Anweisung ist unzulässig, da der Bereich in einer **UNLOCK**-Anweisung mit dem Bereich in den entsprechenden **LOCK**-Anweisungen exakt übereinstimmen muß (es wird kein Fehler angezeigt, aber die Anweisungen erzeugen unvorhersehbare Resultate):

```
LOCK #1, 1 TO 4
LOCK #1, 5 TO 8
UNLOCK #1, 1 TO 8
```

Der folgende Programmausschnitt eröffnet eine Datei und erlaubt einem Benutzer die Sperrung eines einzelnen Satzes, bevor er die Informationen in diesem Satz aktualisiert. Wenn der Benutzer damit fertig ist, gibt das Programm den gesperrten Satz wieder frei. (Die Freigabe von gesperrten Sätzen erlaubt anderen Prozessen den Gebrauch der Daten in der Datei.)

```
TYPE KontoSatz
   Zahlender AS STRING*15
   Adresse AS STRING*20
   Ort AS STRING*20
   Schuld AS SINGLE
END TYPE
DIM KundenSatz AS KontoSatz
OPEN "MONITOR" SHARED AS #1 LEN = LEN(KundenSatz)
DO
   CLS:LOCATE 10,10
   INPUT "Kundennummer?   #"; Nummer%
   ' Sperre den aktuellen Satz, damit ein anderer
   ' Prozeß ihn nicht ändert, während wir ihn
   ' benutzen.
LOCK #1, Nummer%
   GET #1, Nummer%
   LOCATE 11,10: PRINT "Kunde: ";KundenSatz.Zahlender
   LOCATE 12,10: PRINT "Adresse: ";KundenSatz.Adresse
   LOCATE 13,10: PRINT "Schuldet momentan: DM    ";
                 PRINT KundenSatz.Schuld
   LOCATE 15,10: PRINT "Wechselgeld  (+ oder -)";
                 INPUT ":", WechselGeld!
   KundenSatz.Schuld=KundenSatz.Schuld+WechselGeld!
   PUT #1, Nummer%
' Hebe Sperre für Satz auf.
UNLOCK #1, Nummer%
LOCATE 17,10
PRINT "Nochmals aktualisieren"
INPUT "?", Fortfahren$
Aktualisieren$ = UCASE$(LEFT$(Fortfahren$,1))
LOOP WHILE Aktualisieren$="J"
```

LOF-Funktion

Funktion

Gibt die Länge der genannten Datei in Bytes an.

Syntax

LOF (*Dateinummer*)

Anmerkungen

Das Argument *Dateinummer* ist die Nummer, die in der **OPEN**-Anweisung benutzt wurde.

Wenn eine Datei (beliebiger Modus) geöffnet wurde, gibt **LOF** die Größe der Datei in Bytes an.

LOF kann nicht mit den BASIC-Geräten **SCRN:**, **KYBD:**, **CONS:** und **LPT***n***:** benutzt werden. Wenn **LOF** mit einem Gerät benutzt wird, welches mit der **OPEN COM**-Anweisung als eine Datei geöffnet wurde, gibt die **LOF**-Funktion die Anzahl der freien Bytes im Ausgabepuffer an.

Beispiel

Siehe Beispiel zur Anweisung **GET**.

LOG-Funktion

Funktion

Gibt den natürlichen Logarithmus eines numerischen Ausdrucks an.

Syntax

LOG(*n*)

Anmerkungen

Der numerische Ausdruck *n* muß größer als Null sein.

Der natürliche Logarithmus ist der Logarithmus zur Basis *e*. Die Konstante *e* ist ungefähr gleich 2,718282.

Die Funktion **LOG** berechnet den natürlichen Logarithmus mit einfacher Genauigkeit, solange das Argument *n* nicht ein Wert doppelter Genauigkeit ist. In diesem Fall wird **LOG** mit doppelter Genauigkeit berechnet.

Sie können den Logarithmus zur Basis 10 berechnen, indem Sie den natürlichen Logarithmus der Zahl durch den Logarithmus von 10 teilen. Die folgende **FUNCTION** berechnet den Logarithmus zur Basis 10.

```
FUNCTION Log10(X) STATIC
   Log10=LOG(X)/LOG(10.#)
END FUNCTION
```

Beispiele

Das folgende Beispiel gibt zuerst den Wert von *e* an und anschließend den natürlichen Logarithmus von *e* zur ersten, zweiten und dritten Potenz.

```
PRINT EXP (1),
FOR I = 1 TO 3
    PRINT LOG(EXP(1)^I),
NEXT
```

Ausgabe

```
2.718282    1    2    3
```

LPOS-Funktion

Funktion

Gibt die aktuelle Druckkopfposition des Zeilendruckers im Druckerpuffer an.

Syntax

LPOS(*n*)

Anmerkungen

Das Argument *n* ist der Index des zu prüfenden Druckers. Beispielsweise wird `LPT1:` mit `LPOS(1)`, `LPT2:` mit `LPOS(2)` usw. geprüft.

LPOS gibt nicht unbedingt die physische Position des Druckkopfes an, weil sie Tabulator-Zeichen nicht berechnet. Manche Drucker können außerdem Zeichen puffern.

Beispiel

Das nachstehende Programm fordert den Benutzer zur Eingabe von Vereinsnamen und den Namen der Spieler jedes Vereins auf. Anschließend druckt es die Namen der Spieler und ihrer Vereine auf dem Drucker aus.

```
LPRINT"Vereinsmitglieder"; TAB(76); "VEREIN" : LPRINT
INPUT "Wieviele Vereine"; VEREINE
INPUT "Wieviele Spieler pro Verein";SPV
PRINT
FOR V = 1 TO VEREINE
    INPUT "Vereinsname: ",VEREIN$
    FOR S = 1 TO SPV
        PRINT" Geben Sie den Namen des Spielers ein";
        INPUT ":", SPIELER$
        LPRINT SPIELER$;
        IF S < SPV THEN
            IF LPOS(0) > 55 THEN    'Schreibe eine neue
                                    'Zeile, wenn
                                    'Druckkopf Zeile 55
                                    'überschritten hat;
                LPRINT : LPRINT "      ";
            ELSE
                LPRINT ", ";      'andernfalls schreibe
                                  'ein Komma
            END IF
        END IF
    NEXT S
    LPRINT STRING$(80-LPOS(0)- LEN(VEREIN$),".");_
           VEREIN$
NEXT V
```

LPRINT-, LPRINT USING-Anweisungen

Funktion

Drucken Daten auf dem Drucker **LPT1:** aus.

Syntax 1

LPRINT [*Ausdrucksliste*][{; | ,}]

Syntax 2

LPRINT USING *Formatierungs-Zeichenkette*; *Ausdrucksliste* [{; | ,}]

Anmerkungen

Diese Anweisungen funktionieren genauso wie die Anweisungen **PRINT** und **PRINT USING**, mit dem Unterschied, daß die Ausgabe an den Zeilendrucker geleitet wird und die Option *Dateinummer* nicht erlaubt ist.

LPRINT geht von 80 Zeichen pro Druckzeile aus. Diese Zeilenlänge kann mit der Anweisung **WIDTH LPRINT** geändert werden.

Warnung Da die Anweisung **LPRINT** den Drucker **LPT1** benutzt, sollten Sie **LPRINT** nicht in einem Programm verwenden, das eine Anweisung **OPEN LPT1** enthält. Die gemeinsame Verwendung dieser beiden Anweisungen führt zu nicht vorhersehbaren Ergebnissen.

Vergleichen Sie auch

PRINT, PRINT USING, WIDTH

LSET-Anweisung

Funktion

Bringt die Daten aus dem Speicher in einen Direktzugriffs-Dateipuffer (als Vorbereitung auf eine **PUT**-Anweisung) oder ordnet den Wert einer Zeichenkette in einer Zeichenkettenvariablen linksbündig an.

Syntax

LSET *Zeichenkettenvariable* = *Zeichenkettenausdruck*

Anmerkungen

Die *Zeichenkettenvariable* ist normalerweise ein in einer **FIELD**-Anweisung definiertes Direktzugriffsspeicher-Dateifeld, obwohl sie jede beliebige Zeichenkettenvariable sein kann. Der *Zeichenkettenausdruck* ist der Wert, der der Variablen zugewiesen wird.

Wenn *Zeichenkettenausdruck* weniger Bytes erfordert als *Zeichenkettenvariable* in der **FIELD**-Anweisung zugewiesen wurden, ordnet die **LSET**-Funktion die Zeichenkette in dem Feld linksbündig an (**RSET** ordnet die Zeichenkette rechtsbündig an). Die zusätzlichen Stellen werden mit Leerzeichen aufgefüllt. Falls die Zeichenkette zu lang für das Feld ist, schneiden sowohl **LSET** als auch **RSET** Zeichen von rechts ab. Numerische Werte müssen in Zeichenketten umgewandelt werden, bevor sie mit den Anweisungen **LSET** oder **RSET** bündig angeordnet werden können.

Hinweis Sie können **LSET** oder **RESET** außerdem bei einer nicht mit **FIELD** definierten Zeichenkettenvariablen verwenden, um eine Zeichenkette in einem angegebenen Feld linksbündig oder rechtsbündig anzuordnen. Beispielsweise ordnen die Programmzeilen

```
A$=SPACE$(20)
RSET A$=N$
```

die Zeichenkette `N$` in einem 20-Zeichen-Feld rechtsbündig an. Dies kann für die Formatierung von Druckausgaben sehr nützlich sein.

Sie können LSET benutzen, um eine Verbundvariable (Record-Variable) einer anderen zuzuweisen. Das folgende Beispiel kopiert den Inhalt von `VerbZwei` nach `VerbEins`.

```
TYPE ZweiZeichenkette
    Zeikfeld AS STRING * 2
END TYPE
TYPE DreiZeichenkette
    Zeikfeld AS STRING * 3
END TYPE
    DIM VerbEins AS ZweiZeichenkette, VerbZwei AS
    DreiZeichenkette
.
.
.
LSET VerbEins = VerbZwei
```

Beachten Sie, daß **LSET** zur Zuweisung von Verbundvariablen verschiedenen Typs benutzt wird. Verbundvariablen des gleichen Typs können durch **LET** zugewiesen werden. Da außerdem `VerbEins` nur 2 Bytes lang ist, werden nur 2 Bytes von `VerbZwei` kopiert. **LSET** kopiert nur die Anzahl der Bytes von der kürzeren der beiden Verbundvariablen.

Vergleichen Sie auch

LET; MKD$, MKI$, MKL$, MKS$; PUT (Datei E/A); RSET

Beispiel

Die erste Zeile des folgenden Beispiels wandelt die numerische Variable einfacher Genauigkeit `BETRAG` in eine 4-Byte-Zeichenkette um und speichert diese linksbündig in `A$`. Die zweite Zeile wandelt die ganzzahlige numerische Variable `ZAEHL%` in eine 2-Byte-Zeichenkette um und speichert diese rechtsbündig in `D$`.

```
LSET A$ = MKS$(BETRAG)
REST D$ = MKI(ZAEHL%)
```

LTRIM$-Funktion

Funktion

Gibt eine Kopie einer Zeichenkette an, deren führende Leerzeichen entfernt sind.

Syntax

LTRIM$ (*Zeichenkettenausdruck*)

Anmerkungen

Zeichenkettenausdruck kann jeder Zeichenkettenausdruck sein.

Vergleichen Sie auch

RTRIM$

Beispiel

Dieses Programm kopiert eine Datei in eine neue Datei und entfernt alle führenden und nachfolgenden Leerzeichen.

```
CLS
' Eingabe der Dateinamen
INPUT "Geben Sie Namen der Eingabedatei an:",EinDatei$
INPUT "Geben Sie Namen der Ausgabedatei an:",AusDatei$
OPEN EinDatei$ FOR INPUT AS #1
OPEN AusDatei$ FOR OUTPUT AS #2
' Lese, beschneide und schreibe jede Zeile.
DO WHILE NOT EOF (1)
   LINE INPUT #1,ZeileIn$
   ' Entferne vor- und nachstehende Leerzeichen.
   ZeileIn$=LTRIM$(RTRIM$(ZeileIn$))
   PRINT #2, ZeileIn$
LOOP
CLOSE #1,#2
END
```

MID$-Funktion

Funktion

Gibt eine Teilzeichenkette einer Zeichenkette an.

Syntax

MID$ (*Zeichenkettenausdruck, Beginn*[, *Länge*])

Anmerkungen

Die **MID$**-Funktion hat folgende Argumente:

Argument	*Beschreibung*
Zeichenketten-ausdruck	Der Zeichenkettenausdruck, aus dem ein Teilzeichenkettenausdruck herausgenommen wird. Dies kann jeder beliebige Zeichenkettenausdruck sein.
Beginn	Die Zeichenposition in *Zeichenkettenausdruck*, an der der Teilzeichenkettenausdruck beginnt.
Länge	Die Anzahl der herauszunehmenden Zeichen.

Die Argumente *Beginn* und *Länge* müssen im Bereich von 1 bis 32.767 liegen. Wenn *Länge* nicht angegeben wird, oder wenn weniger als *Länge* Zeichen rechts vom *Beginn*-Zeichen stehen, gibt die Funktion **MID$** alle rechts vom *Beginn*-Zeichen stehenden Zeichen aus.

Falls *Beginn* größer ist als die Anzahl der Zeichen in *Zeichenkettenausdruck*, gibt **MID$** eine Null-Zeichenkette aus.

Die Anzahl der Zeichen in *Zeichenkettenausdruck* kann mit der Funktion **LEN** ermittelt werden.

Vergleichen Sie auch

LEFT$, LEN, MID$-Anweisung, RIGHT$

Beispiel

Das folgende Beispiel wandelt binäre Zahlen in Dezimalzahlen um. Es werden Ziffern aus der binären Zahl (Eingabe als Zeichenkette) unter Verwendung von **MID$** ausgegliedert.

```
INPUT "Binärzahl = ",Bin$      'Eingabe Binärzahl als
                               'Zeichenkette
Laenge = LEN(Bin$)             'Stelle Länge der
                               'Zeichenkette fest
Dezimal = 0
FOR K = 1 TO Laenge
   'Stelle die einzelnen Ziffern der Zeichenkette von
   'links nach rechts fest.
   Ziffer$ = MID$(Bin$,K,1)
   'Überprüfe, ob die Ziffer gültig ist.
```

```
   IF Ziffer$="0" OR Ziffer$="1" THEN
      'Wandle Ziffern-Zeichen in Zahlen um:
      Dezimal = 2*Dezimal + VAL(Ziffer$)
   ELSE
      PRINT "Fehler - ungültige Binär-Ziffer ";Ziffer$
      EXIT FOR
   END IF
NEXT
PRINT "Dezimalzahl =" Dezimal
```

Ausgabe

Binärzahl = **10110**

Dezimalzahl = 22

MID$-Anweisung

Funktion

Ersetzt einen Teil einer Zeichenkettenvariablen durch eine andere Zeichenkette.

Syntax

MID$ (*Zeichenkettenvariable*, *Beginn*[, *Länge*]) = *Zeichenkettenausdruck*

Anmerkungen

Die Anweisung **MID$** hat folgende Argumente:

Argument	*Beschreibung*
Zeichenkettenvariable	Die Zeichenkettenvariable, die verändert wird.
Beginn	Ein numerischer Ausdruck, der die Position in *Zeichenkettenvariable* angibt, an der die Veränderung beginnen soll.

Argument	***Beschreibung***
Länge	Die Länge der Zeichenkette, die ersetzt wird. Die *Länge* ist ein numerischer Ausdruck.
Zeichenkettenausdruck	Der Zeichenkettenausdruck, der einen Teil der *Zeichenkettenvariablen* ersetzt.

Die Argumente *Beginn* und *Länge* sind ganzzahlige Ausdrücke. Das Argument *Zeichenkettenvariable* ist eine Zeichenkettenvariable, aber *Zeichenkettenausdruck* kann eine Zeichenkettenvariable, Zeichenkettenkonstante oder ein Zeichenkettenausdruck sein.

Die wahlfreie *Länge* bezieht sich auf die Anzahl der Zeichen aus *Zeichenkettenausdruck*, die in der Ersetzung benutzt werden. Falls *Länge* nicht angegeben wird, werden alle Zeichen von *Zeichenkettenausdruck* benutzt. Ganz gleich, ob *Länge* angegeben wird oder nicht, geht die Ersetzung der Zeichen jedoch niemals über die ursprüngliche Länge von *Zeichenkettenvariable* hinaus.

Vergleichen Sie auch

MID$-Funktion

Beispiel

Das folgende Beispiel benutzt die **MID$**-Anweisung, um ein Zeichen aus einer Zeichenkette zu erhalten.

```
Test = "Paris, Frankreich"
PRINT Test$
MID$(Test$,8)="Texas      "
PRINT Test$
```

Ausgabe

```
Paris, Frankreich
Paris, Texas
```

MKD$-, MKI$-, MKL$-, MKS$-Funktionen

Funktion

Wandelt numerische Werte in Zeichenkettenwerte um.

Syntax

MKI$ (*ganzzahliger Ausdruck*)
MKS$ (*Ausdruck einfacher Genauigkeit*)
MKL$ (*langganzzahliger Ausdruck*)
MKD$ (*Ausdruck doppelter Genauigkeit*)

Anmerkungen

Die **MKI$-**, **MKS$-**, **MKL$-** und **MKD$-**Funktionen werden in Verbindung mit den **FIELD-** und den **PUT-**Anweisungen benutzt, um reelle Zahlen in eine Direktzugriffsdatei zu schreiben. Die Funktionen nehmen numerische Ausdrücke und wandeln sie in Zeichenketten um. Diese können in Zeichenketten, die in der **FIELD-**Anweisung definiert wurden, gespeichert werden. Diese Funktionen sind die Umkehrungen von **CVI**, **CVS**, **CVL** und **CVD**:

Funktion	*Beschreibung*
MKI$	Wandelt eine Ganzzahl in eine Zwei-Byte-Zeichenkette um.
MKS$	Wandelt einen Wert einfacher Genauigkeit in eine Vier-Byte-Zeichenkette um.
MKL$	Wandelt einen langganzzahligen Wert in eine Vier-Byte-Zeichenkette um.
MKD$	Wandelt einen Wert doppelter Genauigkeit in eine Acht-Byte-Zeichenkette um.

Hinweis BASIC-Verbundvariablen bieten eine effizientere und komfortablere Möglichkeit, Direktzugriffsdateien zu schreiben und zu lesen. In Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in QuickBASIC: Ausgewählte Themen* finden Sie weitere Informationen.

Vergleichen Sie auch

CVI, CVS, CVL, CVD; FIELD; PUT

Beispiel

Siehe Beispiel für die Anweisungen **CVI, CVS, CVL,CVD.**

MKDIR-Anweisung

Funktion

Erstellt ein neues Verzeichnis.

Syntax

MKDIR *Verzeichnisname*

Anmerkungen

Verzeichnisname ist ein Zeichenkettenausdruck, der den Namen des zu erstellenden Verzeichnisses angibt. *Verzeichnisname* muß eine Zeichenkette von weniger als 128 Zeichen sein.

Die Anweisung **MKDIR** funktioniert wie der DOS-Befehl **MKDIR**; die Syntax kann in BASIC jedoch nicht wie in DOS auf **MD** abgekürzt werden.

Vergleichen Sie auch

CHDIR, RMDIR

Beispiel

Der folgende Programmausschnitt erstellt ein neues Verzeichnis (falls dieser noch nicht existiert) und kopiert Dateien in dieses Verzeichnis.

```
ON ERROR GOTO FehlerBehandlung
PRINT "Dieses Programm erstellt ein neues ";
PRINT "Verzeichnis namens MONATE";
```

```
PRINT "in diesem Verzeichnis, und erstellt ";
PRINT "dann Dateien in dem Verzeichnis";
MKDIR "MONATE"
DO
    INPUT "Dateiname"; Datei$
    IF Datei$ = "" THEN END
    OPEN "MONATE\"+ Datei$ FOR OUTPUT AS #1
    .
    .
    .
    CLOSE #1
LOOP

FehlerBehandlung:
    'Fehler 75 bedeutet, daß das Verzeichnis MONATE
    'bereits vorhanden ist.
    IF ERR = 75 THEN RESUME NEXT
    ON ERROR GOTO 0
```

MKSMBF$-, MKDMBF$-Funktionen

Funktion

Wandelt eine IEEE-Format-Zahl in eine Zeichenkette um, die eine Microsoft-Binär-Format-Zahl enthält.

Syntax

MKSMBF$ (*Ausdruck einfacher Genauigkeit*)
MKDMBF$ (*Ausdruck doppelter Genauigkeit*)

Anmerkungen

Diese Funktionen werden benutzt, um reelle Zahlen unter Verwendung des Microsoft Binär-Formats in Direktzugriffs-Dateien zu schreiben. Diese Funktionen sind besonders nützlich zur Pflege von Datendateien, die mit älteren BASIC-Versionen erstellt wurden.

Die **MKSMBF$**- und **MKDMBF$**-Funktionen wandeln reelle Zahlen im IEEE-Format in Zeichenketten um, so daß sie in Direktzugriffsdateien geschrieben werden können.

Um eine reelle Zahl im Microsoft-Binär-Format in eine Direktzugriffsdatei zu schreiben, wandeln Sie die Zahl unter Benutzung von **MKSMBF$** (für einfache Genauigkeit) oder von **MKDMBF$** (für doppelte Genauigkeit) um. Dann speichern Sie das Ergebnis in das entsprechende Feld (definiert in der **FIELD**-Anweisung) und schreiben den Satz unter Verwendung der **PUT**-Anweisung in die Datei.

Beispiel

Das folgende Beispiel benutzt **MKSMBF$**, um Werte in einer Datei als Zahlen im Microsoft-Binär-Format zu schreiben.

```
' Lies einen Namen und ein Testergebnis von der
' Tastatur. Speichere als Satz in einer
' Direktzugriffsdatei. Ergebnisse werden als Werte mit
' einfacher Genauigkeit im Microsoft-Binär-Format
' herausgeschrieben.
TYPE Puffer
    NamenFeld AS STRING * 20
    ErgebnisFeld AS STRING * 4
END TYPE
DIM SatzPuffer AS Puffer
OPEN "TESTDAT.DAT" FOR RANDOM AS #1 _
           LEN=LEN(SatzPuffer)
PRINT "Geben Sie Namen und Ergebnisse ein, einen ";
PRINT "Namen und ein Ergebnis pro Zeile."
PRINT "Ende der Eingabe nach END, 0."
INPUT NameEin$, Ergeb
I=0
' Lies Paare von Namen und Ergebnissen von der
' Tastatur bis der Name END ist.
DO WHILE UCASE$(NameEin$) <> "END"
   I=I+1
   SatzPuffer.NamenFeld=NameEin
   ' Übertrage das Ergebnis in eine Zeichenkette.
   SatzPuffer.ErgebnisFeld=MKSMBF$(Ergeb)
   PUT #1,I, SatzPuffer
   INPUT NameEin$, Ergeb
LOOP
PRINT I;" Sätze geschrieben."
CLOSE #1
```

NAME-Anweisung

Funktion

Ändert den Namen einer Disketten-/Plattendatei.

Syntax

NAME *alter Dateiname* **AS** *neuer Dateiname*

Anmerkungen

Die Anweisung **NAME** ähnelt dem DOS-Befehl **RENAME**. Zudem kann **NAME** eine Datei von einem Verzeichnis in ein anderes bewegen.

Die Argumente *alter Dateiname* und *neuer Dateiname* sind Zeichenkettenausdrücke, die einen Dateinamen und einen optionalen Pfad enthalten. Wenn der Pfad in *neuer Dateiname* sich von dem Pfadnamen in *alter Dateiname* unterscheidet, bewegt die **NAME**-Anweisung die Datei und gibt ihr einen neuen Namen.

Alter Dateiname muß bereits existieren, und *neuer Dateiname* darf noch nicht existieren.

Beide Dateien müssen sich auf demselben Laufwerk befinden. Die Verwendung von **NAME** mit unterschiedlichen Ziellaufwerken in dem alten und dem neuen Dateinamen führt zu der Fehlermeldung `Umbenennen zwischen Disketten`.

Nach einer **NAME**-Anweisung existiert die Datei immer noch auf derselben Diskette und in demselben Diskettenbereich, jedoch hat sie nun einen neuen Namen.

Sie können **NAME** nicht benutzen, um Verzeichnisse umzubenennen.

Wenn Sie **NAME** für eine offene Datei benutzen, führt das zu der Laufzeit-Fehlermeldung `Datei bereits geöffnet`. Sie müssen eine offene Datei vor dem Umbenennen schließen.

Beispiele

Die folgenden Anweisungen zeigen die Anwendung von **NAME** mit und ohne Pfadangabe.

```
'Ändert den Namen der Datei KONTO in HAUPTB um.
NAME "KONTO" AS "HAUPTB"

'Bewegt die Datei KUNDEN aus dem Verzeichnis X in das
'Verzeichnis \XYZ\P:
NAME "\X\KUNDEN" AS "\XYZ\P\KUNDEN"
```

OCT$-Funktion

Funktion

Gibt eine Zeichenkette an, die den Oktalwert eines numerischen Arguments darstellt.

Syntax

OCT$ (*Numerischer Ausdruck*)

Anmerkungen

Numerischer Ausdruck kann jeder Typ sein.

Numerischer Ausdruck wird auf eine Ganzzahl oder lange Ganzzahl gerundet, bevor er von der **OCT$**-Funktion ausgewertet wird.

Vergleichen Sie auch

HEX$

Beispiel

Die folgende Zeile gibt die oktale Darstellung von 24 an:

```
PRINT OCT$(24)
```

Ausgabe

```
30
```

ON ERROR-Anweisung

Funktion

Aktiviert die Fehlerbehandlung und gibt die erste Zeile der Fehlerbehandlungsroutine an.

Syntax

ON ERROR GOTO *Zeile*

Anmerkungen

Das Argument *Zeile* ist die Zeilenmarke oder die Zeilennummer der ersten Zeile im Fehlerbehandlungscode. Diese Zeile muß im Modul-Ebenen-Code definiert sein.

Die Zeilennummer 0 schaltet die Fehlerbehandlung aus und legt nicht die Zeile 0 als Anfang des Codes fest. Nachfolgende Fehler geben eine Fehlermeldung aus und halten das Programm an.

Sobald Fehlerbehandlung aktiviert ist, verursacht jeder verfolgbare Fehler einen Sprung in die festgelegte Fehlerbehandlungsroutine. Im Anhang D, "Fehlermeldungen", finden Sie eine vollständige Liste der Fehlermeldungen.

Wird innerhalb einer Fehlerbehandlung eine **ON ERROR**-Anweisung mit der Zeilennummer 0 ausgeführt, hält das Programm an und schreibt die Fehlermeldung für den Fehler, der die Verfolgung verursacht hat. Dies bietet einen komfortablen Weg, ein Programm als Antwort auf Fehler anzuhalten, die nicht von der Fehlerroutine verarbeitet werden können.

Beachten Sie, daß eine Fehlerbehandlungsroutine keine Unterroutine, **SUB**, **DEF FN** oder **FUNCTION** ist. Eine Fehlerbehandlungsroutine ist ein Block von Code, der durch eine einleitende Zeilennummer oder Zeilenmarke markiert ist.

Hinweis In einer Fehlerbehandlungsroutine wird keine Fehlerverfolgung vorgenommen. Fehler während der Ausführung einer Fehlerbehandlungsroutine halten das Programm nach Ausgabe einer Meldung an.

Unterschiede zu BASICA

Das Kompilieren von der **bc**-Befehlszeile aus verlangt zusätzliche Optionen, wenn Sie **ON ERROR** benutzen. BASICA verlangt keine zusätzlichen Optionen. Wenn ein Programm **ON ERROR GOTO**- oder **RESUME**-Anweisungen enthält, müssen Sie die On Error (/e)-Compileroption verwenden, wenn Sie von der **bc**-Befehlszeile aus kompilieren. Wenn ein Programm **RESUME, RESUME NEXT** oder **RESUME 0** enthält, müssen Sie die Resume Next (/x)-Compileroption benutzen, wenn Sie von der **bc**-Befehlszeile aus kompilieren. Wenn Sie in der QuickBASIC-Umgebung arbeiten, sind keine Compileroptionen erforderlich.

Vergleichen Sie auch

ERR, ERL, ERROR, RESUME

Beispiel

Das folgende Programm erhält einen Dateinamen vom Benutzer und zeigt die Datei auf dem Bildschirm an. Wenn die Datei nicht geöffnet werden kann, verfolgt eine Fehlerbehandlungsroutine den Fehler und startet das Programm nochmals mit der Eingabeaufforderung für den Dateinamen.

```
DEFINT A-Z
' Lege die Fehlerbehandlungsroutine fest.
ON ERROR GOTO FehlerBehandlung
CLS
INPUT "Geben Sie die anzuzeigende Datei an: ",DatNam$
' Öffne die Datei.
OPEN DatNam$ FOR INPUT AS #1
' Zeige die Datei auf dem Bildschirm.
DO WHILE NOT EOF(1)
   LINE INPUT #1, Zeile$
   PRINT Zeile$
LOOP
END
' Fehlerbehandlungsroutine behandelt nur
' "Datei nicht gefunden".
' Bricht bei jedem anderen Fehler ab.
CONST DATEINICHTGEFUNDEN=53
FehlerBehandlung:
   IF ERR=DATEINICHTGEFUNDEN THEN
      ' Lies einen anderen Dateinamen.
   PRINT "Datei " UCASE$(DatNam$) " nicht gefunden."
   PRINT "Geben Sie die anzuzeigende Datei an";
   INPUT ":", DatNam$
   RESUME
ELSE
   ' Ein anderer Fehler. Schreibe Meldung und breche
   ' ab.
   PRINT "Nicht abgefangener Fehler - -";ERR
   ON ERROR GOTO 0
END IF
```

ON *Ereignis*-Anweisung

Funktion

Markiert die erste Zeile einer Ereignisverfolgungsunterroutine.

Syntax

ON *Ereignis* **GOSUB**{*Zeilennummer* | *Zeilenmarke*}

Anmerkungen

Die Anweisung **ON** *Ereignis* gibt Ihnen die Möglichkeit, eine Unterroutine festzulegen, die immer dann ausgeführt wird, wenn ein Ereignis auf einem bestimmten Gerät vorliegt. Die folgende Liste beschreibt die Teile der Anweisung:

Teil	***Beschreibung***
Ereignis	Bezeichnet das Ereignis, das zur Verzweigung in eine Ereignisverfolgungsunterroutine führt. Ein Ereignis ist ein Zustand auf einem bestimmten Gerät. Weiter unten finden Sie Informationen zu bestimmten Geräten.
Zeilennummer *Zeilenmarke*	Die Nummer oder die Marke in der ersten Zeile der Ereignisverfolgungsunterroutine. Diese Zeile muß sich im Modul-Ebenen-Code befinden.

Die Zeilennummer 0 schaltet die Ereignisverfolgung aus und bezeichnet nicht etwa den Beginn der Unterroutine.

Die folgende Liste beschreibt die Ereignisse, die verfolgt werden können:

Ereignis	***Beschreibung***
COM(*n*)	Verzweigt zu einer Unterroutine, wenn Zeichen von einem Kommunikationsanschluß empfangen werden. Der ganzzahlige Ausdruck *n* bezeichnet einen der seriellen Anschlüsse, entweder 1 oder 2.
KEY(*n*)	Verzweigt zu der Unterroutine, wenn eine bestimmte Taste betätigt wird. Der ganzzahlige Ausdruck *n* ist die Nummer einer Funktionstaste, Richtungstaste oder einer benutzerdefinierten Taste. In Anhang A, "ASCII-Zeichencodes und Tastaturabfragecodes", und unter der Erklärung zu der Anweisung **KEY** finden Sie weitere Informationen zu diesen Werten.

Ereignis	*Beschreibung*
PEN	Verzweigt zu der Unterroutine, wenn ein Lichtstift aktiviert ist.
PLAY(*Warteschlangengrenze*)	Verzweigt, wenn die Anzahl der Noten in dem Musikpuffer von *Warteschlangengrenze* zu *Warteschlangengrenze* -1 geht. Der Wert *Warteschlangengrenze* ist eine ganzzahliger Ausdruck von 1 bis 32. Eine **PLAY**-Ereignisverfolgungsunterroutine kann in Verbindung mit einer **PLAY**-Anweisung benutzt werden, um während des gesamten Programmablaufes eine Hintergrundmusik zu spielen.
STRIG(*n*)	Verzweigt zu der Ereignisverfolgungsunterroutine, wenn eine Joystick-Taste betätigt wird. Der ganzzahlige Ausdruck *n* bezeichnet die einzelnen Joystick-Tasten: 0 steht für die untere Taste und 4 steht für die obere Taste des ersten Joysticks; 2 steht für die untere Taste und 6 steht für die obere Taste des zweiten Joysticks.
TIMER(*n*)	Verzweigt zu der Unterroutine, wenn *n* Sekunden vergangen sind. Der ganzzahlige Ausdruck *n* kann im Bereich von 1 bis 86.400 (1 Sekunde bis 24 Stunden) liegen.

Die Anweisung **ON** *Ereignis* legt nur den Anfang einer Ereignisverfolgungsunterroutine fest. Eine Reihe anderer Anweisungen legt fest, ob eine Unterroutine aufgerufen wird oder nicht. Diese Anweisungen schalten die Ereignisverfolgung ein oder aus und entscheiden, wie die Ereignisse behandelt werden, wenn die Ereignisverfolgung ausgeschaltet ist. Die folgende Liste beschreibt diese Anweisungen im allgemeinen. Unter den Beschreibungen der einzelnen Anweisungen in diesem Buch finden Sie detailliertere Informationen.

Ereignis	*Beschreibung*
Ereignis **ON**	Aktiviert die Ereignisverfolgung. Ereignisverfolgungen finden nur statt, wenn zuvor eine *Ereignis* **ON**-Anweisung ausgeführt wurde.
Ereignis **OFF**	Schaltet die Ereignisverfolgung aus. Solange keine weitere *Ereignis* **ON**-Anweisung ausgeführt wird, findet keine Ereignisverfolgung statt. Wenn Ereignisverfolgung ausgeschaltet ist, wird das Auftreten eines Ereignisses ignoriert.

Ereignis	***Beschreibung***
Ereignis **STOP**	Verhindert Ereignisverfolgung bis zur nächsten Ausführung einer *Ereignis* **ON**-Anweisung. Das Auftreten eines Ereignisses, während die Ereignisverfolgung ausgeschaltet ist, wird festgehalten und nach der nächsten Ausführung einer *Ereignis* **ON**-Anweisung verarbeitet.

Wenn eine Ereignisverfolgung stattfindet (die Unterroutine wird aufgerufen), führt BASIC automatisch eine *Ereignis* **STOP**-Anweisung aus, die die rekursive Verfolgung unterbindet. Beim Verlassen der Unterroutine mit der Anweisung **RETURN** wird automatisch wieder eine *Ereignis* **ON**-Anweisung ausgeführt, es sei denn, innerhalb der Unterroutine wird explizit eine *Ereignis* **OFF**-Anweisung ausgeführt.

Hinweis Auf Grund der impliziten *Ereignis* **STOP**- und *Ereignis* **ON**-Anweisungen, werden Ereignisse während der Ausführung der Verfolgungsunterroutine festgehalten und verarbeitet, wenn die Verfolgungsroutine endet.

Die Formen **RETURN** *Zeilennummer* oder **RETURN** *Zeilenmarke* der **RETURN**-Anweisung können verwendet werden, um zu einer bestimmten Zeilennummer von der verfolgenden Unterroutine aus zurückzukehren. Benutzen Sie diese Art des Rücksprungs jedoch vorsichtig, da alle **GOSUB**-, **WHILE**- und **FOR**-Anweisungen, die während der Verfolgung aktiv waren, aktiv bleiben. Dies kann Fehler wie `NEXT ohne FOR` verursachen. Darüber hinaus kann, wenn ein Ereignis in einer Prozedur vorkommt, eine **RETURN** *Zeilennummer* oder eine **RETURN** *Zeilenmarke* keine Rückkehr in die Prozedur bewirken - die Zeilennummer oder -marke muß sich im Modul-Ebenen-Code befinden.

Die nächsten drei Abschnitte enthalten zusätzliche Informationen zu den Anweisungen **ON COM, ON KEY** und **ON PLAY**.

Wie Sie ON COM benutzen

Wenn Ihr Programm Daten mit einer asynchronen Datenübertragungsschnittstelle empfängt, kann die BASIC-Befehlszeilen-Option **/c** benutzt werden, um die Größe des Datenpuffers einzustellen. Weitere Informationen finden Sie in Kapitel 9, "Kompilieren und Binden von DOS", in *Lernen und Anwenden von Microsoft QuickBASIC*.

Wie Sie ON KEY benutzen

Tasten werden in der folgenden Reihenfolge verarbeitet:

1. Der Echo-Umschalter des Zeilendruckers wird zuerst abgearbeitet. Das Definieren dieser Taste als benutzerdefinierte Taste verhindert nicht, daß Zeichen nach Betätigung auf dem Zeilendrucker ausgegeben werden.
2. Funktionstasten und Richtungstasten werden anschließend untersucht. Das Definieren einer FUNKTIONS- oder RICHTUNGSTASTE als benutzerdefinierte Taste hat keine Auswirkung, da diese Tasten vorbelegt sind.
3. Zuletzt werden die benutzerdefinierten Tasten untersucht.

Die **ON KEY**-Anweisung kann alle Tasten verfolgen, so auch die UNTBR-TASTE und System-Reset. Dies verhindert das versehentliche Verlassen des Programms oder ein erneutes Starten des Rechners.

Hinweis Wenn eine Taste verfolgt wird, ist das Tastatur-Ereignis zerstört. Danach können Sie die Anweisungen **INPUT** oder **INKEY$** nicht mehr verwenden, um festzustellen, welche Taste die Verfolgung verursachte. Da keine Möglichkeit besteht festzustellen, welche Taste die Verzweigung zu der Verfolgung verursachte, müssen Sie eine Unterroutine für jede Taste schreiben, wenn Sie verschiedene Funktionen den einzelnen Tasten zuweisen wollen.

Wie Sie ON PLAY benutzen

Die folgenden drei Regeln kommen beim Benutzen von **ON PLAY** zur Anwendung:

1. Eine Spielen-Ereignisverfolgung findet nur statt, wenn Musik im Hintergrund spielt. Spielen-Ereignisverfolgungen finden nicht statt, wenn Musik im Vordergrund läuft.
2. Eine Spielen-Ereignisverfolgung findet nicht statt, wenn die Warteschlange der Hintergrundmusik bereits von *Warteschlangengrenze* zu *Warteschlangengrenze* -1 Noten gegangen ist, wenn eine **PLAY ON**-Anweisung ausgeführt wird.
3. Wenn *Warteschlangengrenze* eine große Zahl ist, können Ereignisverfolgungen so häufig erfolgen, daß das Programm verlangsamt wird.

Unterschiede zu BASICA

Falls Sie **bc** von der DOS-Eingabeaufforderung aus benutzen, müssen Sie /v oder /w verwenden, wenn ein Programm **ON *Ereignis***-Anweisungen enthält. Diese Optionen ermöglichen es dem Compiler, richtig zu arbeiten, wenn Unterroutinen zur Ereignisverfolgung in ein Programm eingebunden sind. BASICA erfordert keine zusätzliche Optionen.

Vergleichen Sie auch

COM, KEY(*n*), PEN ON, PLAY ON, RETURN, STRIG ON, TIMER ON

Beispiele

Das folgende Beispiel spielt fortwährend Musik, indem eine Ereignisbehandlungsroutine immer dann aufgerufen wird, wenn der Musikpuffer von 3 auf 2 Noten geht.

```
'  Rufe Unterroutine Hintergrund auf, wenn Musikpuffer
'  von 3 auf 2 Noten geht.
ON PLAY(3) GOSUB Hintergrund
'  Einschalten der Ereignisverfolgung von PLAY
PLAY ON
'  Definiere eine Zeichenkette, die die Melodie
'  enthält.
Lizzie$=" o3 L8 E D+ E D+ E o2 B o3 D C L2 o2 A"
'  Spiel die Melodie zum ersten Mal.
PLAY "MB X" + VARPTR$(Lizzie$)
'  Fortfahren, bis eine Taste betätigt wird.
text$="Beliebige Taste betätigen, um anzuhalten."
LOCATE 2,1 : PRINT text$
DO WHILE INKEY$=""
LOOP
END
'  Ereignisbehandlungsunterroutine PLAY.
Hintergrund:
   ' Zähle hoch und schreibe jedesmal den Zähler.
   Zaehl% = Zaehl% + 1
   LOCATE 1,1 : PRINT "Hintergrund ";Zaehl%;
                PRINT " mal    aufgerufen"
   ' Führe weiteres PLAY aus, um den Puffer zu füllen.
   PLAY "MB X" + VARPTR$(Lizzie$)
   RETURN
```

Das folgende Beispiel zeichnet alle drei Sekunden ein Vieleck mit zufälligem Umriß (drei bis sieben Seiten), zufälliger Größe und zufälligem Ort:

```
SCREEN 1
DEFINT A-Z
DIM X(6), Y(6)
TIMER ON        'Aktiviere TIMER-Ereignisverfolgung
ON TIMER(3) GOSUB Zeichvieleck  'Zeichne alle drei
                                'Sekunden ein neues
                                'Vieleck

PRINT "Irgendeine Taste betätigen, um das";
PRINT " Programm zu beenden"
INPUT "<EINGABETASTE> betätigen, um zu starten",Test$
DO
LOOP WHILE INKEY$=""    'Zum Beenden des Programms eine
                        'Taste betätigen

END
Zeichvieleck:
   CLS         'Lösche altes Vieleck
   N = INT(5*RND + 2)   'N ist Zufallszahl von 2 bis 6
   FOR I = 0 TO N
      X(I) = INT(RND*319)    'Stelle Koordinaten der
      Y(I) = INT(RND*199)    'Ecken des Vielecks fest
   NEXT
   PSET (X(N),Y(N))
   FOR I = 0 TO N
      LINE - (X(I),Y(I)),2    'Zeichne ein neues
                              'Vieleck

   NEXT
   RETURN
```

ON...GOSUB-, ON...GOTO-Anweisungen

Funktion

Verzweigt zu einer von mehreren angegebenen Zeilen, abhängig von dem Wert eines Ausdrucks.

Syntax 1

ON *Ausdruck* **GOTO** {*Zeilennummernliste* | *Zeilenmarkenliste*}

Syntax 2

ON *Ausdruck* **GOSUB** {*Zeilennummernliste* | *Zeilenmarkenliste*}

Anmerkungen

Das Argument *Ausdruck* kann jeder numerische Ausdruck sein (*Ausdruck* wird vor der Auswertung der Anweisung **ON...GOSUB** oder **ON...GOTO** auf eine ganze Zahl gerundet). *Zeilennummernliste* oder *Zeilenmarkenliste* besteht aus einer Liste von Zeilennummern oder Zeilenmarken, die durch Kommata voneinander getrennt werden. Der Wert von *Ausdruck* legt fest, zu welcher Zeile das Programm verzweigt. Wenn der Wert beispielsweise 3 ist, ist die dritte in der Liste angegebene Zeile das Ziel der Verzweigung.

Der Wert von *Ausdruck* sollte größer oder gleich 1 und kleiner oder gleich der Anzahl der Objekte in der Liste sein. Wenn der Wert außerhalb dieses Bereiches liegt, folgt eines der folgenden Ergebnisse:

Wert	*Ergebnis*
Nummer gleich 0 oder größer als die Angaben in der Liste	Kontrolle geht an die nächste BASIC-Anweisung.
Negative Nummer oder größer als 255	Fehlermeldung `Unzulässiger Funktionsaufruf` erscheint.

Sie können Zeilennummern und -marken in der Liste mischen.

Hinweis Die **SELECT CASE**-Anweisung bietet eine leistungsfähigere, komfortablere und flexiblere Möglichkeit, mehrere Verzweigungen zu bearbeiten.

Vergleichen Sie auch

GOSUB, RETURN, SELECT CASE

Beispiel

Der folgende Programmausschnitt bewirkt die Verzweigung der Programmkontrolle zu einer der vier aufgelisteten Unterroutinen, abhängig vom Wert von `ZchWert`:

```
DO
   CLS
   PRINT "1) Teilnehmer der Seminare ausgeben."
   PRINT "2) Die gesamten bezahlten ";
   PRINT "Anmeldegebühren berechnen."
   PRINT "3) Adressenlisten ausgeben."
   PRINT "4) Programmende."
   PRINT : PRINT "Welche Wahl treffen Sie?"
   DO
      Zch$=INKEY$
   LOOP WHILE Zch$=""
   ZchWert = VAL(Zch$)
   IF ZchWert > 0  AND ZchWert < 5 THEN
        ON ZchWert GOSUB Seminare, Gebuehren, _
           Adressen, Ende
   END IF
LOOP
END
.
.
.
```

OPEN-Anweisung

Funktion

Erlaubt eine Eingabe/Ausgabe in eine Datei oder ein Gerät.

Syntax 1

OPEN *Datei* [**FOR** *Modus1*][**ACCESS** *Zugriff*][*Sperrtyp*]**AS**[#]*Dateinummer* [**LEN**=*Satzlänge*]

Syntax 2

OPEN *Modus2*,[#]*Dateinummer*, *Datei*[, *Satzlänge*]

Anmerkungen

Datei ist ein Zeichenkettenausdruck, der ein wahlweises Gerät angibt, gefolgt von einem mit den DOS-Konventionen zur Dateibenennung übereinstimmenden Datei- oder Pfadnamen.

Sie müssen eine Datei eröffnen, bevor irgendeine E/A-Operation mit ihr ausgeführt werden kann. **OPEN** ordnet der Datei oder dem Gerät einen Puffer für E/A zu und bestimmt den Modus, mit dem auf den Puffer zugegriffen werden kann.

Die nächsten zwei Abschnitte beschreiben die beiden Formen der Anweisung.

Erste Form der OPEN-Anweisung

In der ersten Syntax ist *Modus1* einer der folgenden Ausdrücke:

Modus	*Beschreibung*
OUTPUT	Gibt sequentiellen Ausgabemodus an.
INPUT	Gibt sequentiellen Eingabemodus an.
APPEND	Gibt sequentiellen Ausgabemodus an und setzt den Dateizeiger an das Dateiende und die Satznummer auf den letzten Satz der Datei. Eine Anweisung **PRINT #** oder **WRITE #** verlängert dann die Datei (d. h. fügt ihr Sätze hinzu).
RANDOM	Gibt Direktzugriffs-Dateimodus, den Standard-Dateimodus, an. Wenn im **RANDOM**-Modus keine **ACCESS**-Klausel vorhanden ist, werden drei Versuche gemacht, die Datei bei der Ausführung der **OPEN**-Anweisung zu öffnen. Der Zugriff wird in dieser Reihenfolge versucht: **1.** Lesen/Schreiben **2.** Nur Schreiben **3.** Nur Lesen
BINARY	Gibt Binär-Dateimodus an. Im Binärmodus können Sie Informationen an jede Byte-Position einer Datei mit **GET** und **PUT** lesen bzw. schreiben. Im **BINARY**-Modus werden - sofern keine **ACCESS**-Klausel angegeben wird - drei Versuche gemacht, die Datei zu öffnen. Dies geschieht in derselben Weise wie für **RANDOM**-Dateien.

Wenn *Modus1* nicht angegeben wird, wird der standardmäßige Direktzugriffsmodus angenommen.

Der Ausdruck *Zugriff* legt die Operation fest, die mit der geöffneten Datei vorgenommen werden soll. Wenn die Datei bereits von einem anderen Prozeß geöffnet wurde, und der angegebene Zugriffstyp nicht erlaubt ist, erscheint die Fehlermeldung `Zugriff nicht gestattet`. Die **ACCESS**-Klausel funktioniert in einer **OPEN**-Anweisung nur, wenn Sie eine DOS-Version benutzen, die netzwerkfähig (DOS 3.0 oder später) ist. Außerdem müssen Sie das **SHARE.EXE**-Programm starten (oder das Netzwerk-Startprogramm muß es starten), um irgendeine Sperroperation auszuführen. Frühere Versionen von DOS geben die Fehlermeldung `Erweiterte Eigenschaft nicht verfügbar` aus, wenn **ACCESS** mit **OPEN** benutzt wird.

Zugriff kann eines der folgenden Argumente sein:

Zugriffstyp	*Beschreibung*
READ	Öffnet eine Datei nur zum Lesen.
WRITE	Öffnet eine Datei nur zum Schreiben.
READ WRITE	Öffnet eine Datei sowohl zum Lesen als auch zum Schreiben. Dieser Modus ist nur für **RANDOM**- und **BINARY**-Dateien und für Dateien, die für **APPEND** eröffnet wurden, gültig.

Die *Sperrtyp*-Klausel funktioniert in einer Mehrprozessorumgebung und beschränkt den Zugriff von anderen Prozessen auf eine geöffnete Datei. Es gibt folgende Sperrtypen:

Sperrtyp	*Beschreibung*
Standard	Wenn *Sperrtyp* nicht angegeben wird, kann die Datei von diesem Prozeß beliebig oft zum Lesen oder Schreiben geöffnet werden, während anderen Prozessen der Zugriff auf die geöffnete Datei nicht gestattet wird.
SHARED	Jeder Prozeß auf jedem Computer kann aus dieser Datei lesen oder in sie schreiben. Verwechseln Sie den **SHARED**-Sperrtyp nicht mit dem **SHARED**-Attribut in anderen Anweisungen.
LOCK READ	Keinem anderen Prozeß wird ein Lesezugriff auf diese Datei gestattet. Dieser Zugriff wird nur gestattet, wenn kein anderer Prozeß einen vorhergehenden **READ**-Zugriff auf die Datei hat.

Sperrtyp	*Beschreibung*
LOCK WRITE	Keinem anderen Prozeß wird der Schreibzugriff auf diese Datei gestattet. Dieser Zugriff wird nur gestattet, wenn kein anderer Prozeß einen vorhergehenden **WRITE**-Zugriff auf die Datei hat.
LOCK READ WRITE	Keinem anderen Prozeß wird Lese- oder Schreibzugriff auf diese Datei gestattet. Dieser Zugriff wird nur gestattet, wenn ein **READ**- oder **WRITE**-Zugriff nicht bereits einem anderen Prozeß gestattet wurde, oder wenn nicht bereits ein **LOCK READ** oder **LOCK WRITE** vorliegt.

Wenn das **OPEN** durch einen vorhergehenden Prozeß eingeschränkt ist, gibt es unter DOS Fehler 70, `Zugriff nicht gestattet`, aus.

Das Argument *Dateinummer* ist ein ganzzahliger Ausdruck, dessen Wert zwischen 1 und 255 liegt. Wenn ein **OPEN** ausgeführt wird, ist die Nummer mit der Datei verknüpft, solange diese geöffnet ist. Weitere E/A-Anweisungen können diese Nummer verwenden, um sich auf die Datei zu beziehen.

Das Argument *Satzlänge* ist ein ganzzahliger Ausdruck, der die Satzlänge (Anzahl der Zeichen in einem Satz) für Direktzugriffsdateien setzt. Die Standardlänge für Sätze ist 128 Bytes; *Satzlänge* darf 32.767 Bytes nicht überschreiten. Wenn der Dateimodus **BINARY** ist, wird die **LEN**-Klausel ignoriert.

Bei sequentiellen Dateien muß *Satzlänge* nicht einer einzelnen Satzlänge entsprechen, da eine sequentielle Datei Sätze verschiedener Länge haben kann. Wenn *Satzlänge* zum Eröffnen einer sequentiellen Datei verwendet wird, gibt es die Anzahl der Zeichen an, die in dem Puffer zwischenzuspeichern sind, bevor der Puffer auf die Platte geschrieben oder von ihr gelesen wird. Ein größerer Puffer bedeutet weniger Speicherplatz für BASIC, aber schnellere Datei-E/A. Ein kleinerer Puffer bedeutet mehr Platz im Speicher für BASIC, aber langsamere E/A. Die Standard-Puffergröße ist 512 Bytes.

Zweite Form der OPEN-Anweisung

In der zweiten Form der **OPEN**-Syntax ist *Modus2* ein Zeichenkettenausdruck, der mit einem der folgenden Zeichen anfangen muß:

Modus	*Beschreibung*
O	Gibt sequentiellen Ausgabemodus an.
I	Gibt sequentiellen Eingabemodus an.
R	Gibt Modus als Direktzugriffseingabe/-ausgabe an.

Modus	*Beschreibung*
B	Gibt Binär-Dateimodus an.
A	Gibt sequentiellen Ausgabemodus an und setzt den Dateizeiger an das Dateiende und die Satznummer auf den letzten Satz der Datei. Eine Anweisung **PRINT #** oder **WRITE #** verlängert dann die Datei (d. h. fügt ihr Sätze hinzu).

Hinweis Die zweite Form der **OPEN**-Syntax unterstützt keine der Optionen für den Zugriff auf eine Datei oder deren gemeinsame Benutzung wie in der ersten Syntax. Sie wird verwendet, um Kompatibilität mit Programmen herzustellen, die in früheren BASIC-Versionen geschrieben wurden.

BASIC-Geräte

Folgende Geräte werden von BASIC unterstützt und können mit dem Argument *Dateiangabe* benannt und aktiviert werden:

KYBD:

SCRN:

COM*n*:

LPT*n*:

CONS:

Mit dem E/A-System für BASIC-Dateien können Sie die vom Benutzer installierten Geräte benutzen. (Informationen über alphanumerische Geräte finden Sie in Ihrem DOS-Handbuch.)

Alphanumerische Geräte werden genauso wie Dateien geöffnet (in diesem Fall aktiviert) und verwendet. Zeichen werden jedoch von BASIC nicht wie für Plattendateien zwischengespeichert. Die Satzlänge für die Gerätedateien wird auf 1 gesetzt.

BASIC sendet einen Wagenrücklauf nur am Zeilenende. Wenn das Gerät einen Zeilenvorschub benötigt, muß der Treiber ihn liefern. Denken Sie beim Schreiben von Gerätetreibern daran, daß andere BASIC-Benutzer Steuerinformationen lesen und schreiben wollen. Das Lesen und Schreiben von Geräte-Steuerdaten erfolgt durch die Anweisung **IOCTL** und die Funktion **IOCTL$**.

Keines der BASIC-Geräte unterstützt den Binärmodus direkt. Die Druckerschnittstellen (**LPT1:**, **LPT2:**) können durch hinzufügen des **BIN**-Schlüsselwortes im Binärmodus geöffnet werden:

```
OPEN "LPT1:BIN" FOR OUTPUT AS #1
```

Das Öffnen eines Druckers im **BIN**-Modus schließt das Drucken eines Wagenrücklaufes am Ende einer Zeile aus.

Hinweis Im **INPUT**-, **RANDOM**- und **BINARY**-Modus können Sie eine Datei unter verschiedenen Nummern öffnen, ohne diese vorher zu schließen. Im **OUTPUT**- oder **APPEND**-Modus müssen Sie eine Datei schließen, bevor Sie diese wieder mit einer anderen Dateinummer öffnen.

Vergleichen Sie auch

FREEFILE

Beispiele

Das folgende Beispiel öffnet `ADRESSEN.DAT` als Datei Nummer 1 und ermöglicht das Hinzufügen von Daten, ohne daß der bereits vorhandene Inhalt von `ADRESSEN.DAT` gelöscht wird:

```
OPEN "ADRESSEN.DAT" FOR APPEND AS #1
```

Wenn Sie einen Gerätetreiber namens `ROBOTER` schreiben und installieren, kann die **OPEN**-Anweisung folgendermaßen aussehen:

```
OPEN "\GER\ROBOTER" FOR OUTPUT AS #1
```

Um den Drucker für Ausgaben zu aktivieren, können Sie eine der beiden folgenden Zeilen benutzen (die erste Zeile benutzt das BASIC-Gerät `LPT1:`, während die zweite Zeile das DOS-Gerät `LPT1` benutzt):

```
OPEN "LPT1:" FOR OUTPUT AS #1
OPEN "LPT1" FOR OUTPUT AS #1
```

Die nachstehende Anweisung öffnet die Datei `SAETZE` im Direktzugriffsmodus nur zum Lesen. Die Anweisung sperrt die Datei für Schreibzugriffe, erlaubt aber anderen Prozessen das Lesen, solange **OPEN** wirksam ist:

```
OPEN "SAETZE" FOR RANDOM ACCESS READ LOCK WRITE AS #1
```

Das nachstehende Beispiel öffnet die Datei `BESTAND` für Eingabe als Datei Nummer 2:

```
OPEN "I",2,"BESTAND"
```

OPEN COM-Anweisung

Funktion

Aktiviert und initialisiert einen Datenübertragungskanal für E/A.

Syntax

OPEN "COM*n*: *Optionsliste1 Optionsliste2*" [**FOR** *Modus*] **AS** [#]*Dateinummer* [**LEN**=*Satzlänge*]

Anmerkungen

COM*n*: ist der Name des zu aktivierenden Gerätes. Das Argument *n* ist die Nummer eines zulässigen Datenübertragungsgerätes wie **COM1**: oder **COM2**: Die erste Liste der Optionen, *Optionsliste1*, hat folgende Form:

[*Übertragungsrate*][,[*Parität*][,[*Daten*][,[*Stop*]]]]

Die folgende Tabelle beschreibt die möglichen Optionen.

Option	***Beschreibung***
Übertragungsrate	Die Baud-Rate (Baud entspricht "Bits pro Sekunde") des zu aktivierenden Gerätes.
Parität	Die Parität des zu aktivierenden Gerätes. Gültige Einträge für Parität sind: **N** (non = keine), **E** (even = gerade), **O** (odd = ungerade), **S** (space = Leerzeichen) oder **M** (mark = Kennzeichen).
Daten	Die Anzahl der Datenbits pro Byte. Gültige Einträge sind 5, 6, 7 oder 8.
Stop	Die Anzahl der Stopp-Bits. Gültige Einträge sind 1, 1.5 oder 2.

Die Optionen aus dieser Liste müssen in der gezeigten Reihenfolge eingegeben werden; falls Optionen aus *Optionsliste2* gewählt werden, müssen außerdem Kommata als Platzhalter auch dann verwendet werden, wenn keine der Optionen aus *Optionsliste1* gewählt wird. Zum Beispiel:

```
OPEN "COM1: ,,,,CD1500" FOR INPUT AS #1
```

Wenn Sie die Anzahl der Datenbits pro Byte auf 8 setzen, müssen Sie N (keine Parität) angeben. Weil QuickBASIC komplette Bytes (8 Bits) für die Zahlendarstellung benutzt, müssen Sie 8 Datenbits angeben, wenn Sie numerische Daten senden oder empfangen.

Die Wahlmöglichkeiten für *Optionsliste2* werden in der folgenden Liste beschrieben. Das Argument *m* wird in Millisekunden angegeben; der Standardwert für *m* ist 1000.

Option	*Beschreibung*
ASC	Öffnet das Gerät im ASCII-Modus. Im ASCII-Modus werden Tabulatoren zu Leerzeichen erweitert, am Zeilenende werden Wagenrückläufe erzwungen, STRG+Z wird als Dateiende behandelt und das XON/XOFF-Protokoll ist eingeschaltet. Wenn der Kanal geschlossen ist, wird STRG+Z über die RS-232-Leitung gesendet.
BIN	Öffnet das Gerät im Binärmodus. Diese Option ersetzt die Option **LF**. **BIN** wird standardmäßig gewählt, solange nicht **ASC** angegeben wird. Im **BIN**-Modus werden Tabulatoren nicht zu Leerzeichen erweitert, am Zeilenende werden keine Wagenrückläufe erzwungen und STRG+Z wird nicht als Dateiende behandelt. Wenn der Kanal geschlossen ist, wird STRG+Z nicht über die RS-232-Leitung gesendet.
CD[*m*]	Kontrolliert die Zeitüberschreitung auf der Empfangssignalpegel-Leitung (Data Carrier Detect, DCD). Wenn DCD länger als *m* Millisekunden unterdrückt ist, tritt eine Zeitüberschreitung auf.
CS[*m*]	Kontrolliert die Zeitüberschreitung auf der Sendebereitschafts-Leitung (Clear To Send, CTS). Wenn CTS länger als *m* Millisekunden unterdrückt ist (es liegt kein Signal an), tritt eine Zeitüberschreitung auf.
DS[*m*]	Kontrolliert die Zeitüberschreitung auf der DÜE-Betriebsbereitschafts-Leitung (Data Set Ready, DSR). Wenn DSR länger als *m* Millisekunden unterdrückt ist, tritt eine Zeitüberschreitung auf.
LF	Erlaubt das Ausdrucken von Kommunikationsdateien auf einem seriellen Zeilendrucker. Wenn **LF** angegeben wird, wird ein Zeilenvorschubzeichen (0AH) automatisch nach jedem Wagenrücklaufzeichen (0DH) gesendet. Dies schließt den als Ergebnis der Breiteneinstellung gesendeten Wagenrücklauf ein. Beachten Sie, daß **INPUT** und **LINE INPUT**, wenn sie dazu verwendet werden, aus einer mit der Option **LF** geöffneten **COM**-Datei zu lesen, anhalten und den Zeilenvorschub ignorieren, wenn sie einen Wagenrücklauf entdecken.

Option	*Beschreibung*
OP[*m*]	Kontrolliert, wie lange die Anweisung auf ein erfolgreiches Öffnen wartet. Der Parameter *m* ist ein Wert in dem Bereich von 0 bis 65.535, der den Zeitraum in Millisekunden angibt, in dem auf die Aktivierung der Kommunikationsleitungen gewartet wird. Wenn **OP** ohne einen Wert angegeben ist, wartet die Anweisung unbegrenzt. Wenn **OP** ausgelassen ist, wartet **OPEN COM** zehnmal solange, wie der maximale Wert der Zeitüberschreitungwerte von **CD** oder **DS** angibt.
RB[*n*]	Setzt die Größe des Empfangspuffers auf *n* Bytes. Wenn *n* oder die Option ausgelassen ist, wird der aktuelle Wert der Option **/c** auf der QuickBASIC- oder **bc**-Befehlszeile gesetzt werden. Die Standardeinstellung ist 256 Bytes. Die maximale Größe ist 32.767 Bytes.
RS	Unterdrückt die Überprüfung von Sendeteil Einschalten (Request To Send, RTS).
TB[*n*]	Setzt die Größe des Übertragungspuffers auf *n* Bytes. Wenn *n* oder die Option ausgelassen ist, wird der aktuelle Wert verwendet. Die Standardeinstellung ist 128 Bytes.

Die Optionen der obigen Liste können in beliebiger Reihenfolge eingegeben werden. Sie müssen jedoch durch Kommata voneinander getrennt werden. Für **CS**[*m*], **DS**[*m*] und **CD**[*m*] tritt eine Zeitüberschreitung ein, wenn eine Leitung nicht innerhalb von *m* Millisekunden gesetzt ist (es liegt ein Signal an). Wenn *m* gleich 0 ist, so gilt für alle Optionen, daß diese Option ignoriert wird. Wenn die Option **CS** angegeben ist, wird die CTS-Leitung jedesmal überprüft, wenn sich Daten im Übertragungspuffer befinden. Die DSR- und die DCD-Leitungen werden fortwährend auf Zeitüberschreitungen hin überprüft, wenn die entsprechenden Optionen (**DS**, **CD**) angegeben sind.

Das Argument *Modus* ist einer der folgenden Zeichenkettenausdrücke:

Modus	*Beschreibung*
OUTPUT	Gibt sequentiellen Ausgabemodus an.
INPUT	Gibt sequentiellen Eingabemodus an.
RANDOM	Gibt Direktzugriffsmodus an.

Wenn der Ausdruck *Modus* ausgelassen wird, wird der Direktzugriffsmodus für Eingabe/Ausgabe angenommen.

Die *Dateinummer* ist die Nummer, die zur Öffnung der Datei benutzt wird. Die **OPEN COM**-Anweisung muß ausgeführt werden, bevor ein Gerät zur Kommunikation über eine RS-232-Schnittstelle verwendet werden kann.

Wenn das Gerät im **RANDOM**-Modus geöffnet ist, legt die **LEN**-Option die Länge eines zugeordneten Direktzugriffspuffers fest. Der Standardwert für die Länge ist 128. Sie können nur Direktzugriffs-E/A-Anweisungen verwenden, wie **GET** und **PUT**, um das Gerät wie eine Direktzugriffsdatei zu behandeln.

Die **OPEN COM**-Anweisung führt folgende Schritte zur Öffnung eines Kommunikationsgerätes durch:

1. Die Kommunikationspuffer werden zugewiesen und Unterbrechungen (Interrupts) werden aktiviert.
2. Die Betriebsbereitschafts-Leitung (Data Terminal Ready, DTR) wird gesetzt.
3. Wenn eine der Optionen **OP** oder **DS** ungleich Null ist, wartet die Anweisung bis zur angegebenen Zeit darauf, daß die DÜE-Betriebsbereitschafts-Leitung (DSR) gesetzt wird. Wenn eine Zeitüberschreitung auftritt, geht die Verarbeitung zu Schritt 6.
4. Die Leitung Sendeteil Einschalten (RTS) wird gesetzt, wenn die **RS**-Option nicht angegeben ist.
5. Wenn eine der Optionen **OP** oder **CD** ungleich Null ist, wartet die Anweisung bis zur angegebenen Zeit darauf, daß die Empfangssignalpegel-Leitung (DCD) gesetzt wird. Wenn eine Zeitüberschreitung auftritt, geht die Verarbeitung zu Schritt 6. Anderenfalls war **OPEN COM** erfolgreich.
6. Das Öffnen kann aufgrund von Zeitüberschreitungen nicht durchgeführt werden. Die Verarbeitung gibt die Puffer wieder frei, deaktiviert Unterbrechungen (Interrupts) und löscht alle Steuerungsleitungen.

Hinweis Im Vergleich zu den Optionen **CS**, **DS** oder **CD** sollten Sie für die Option **OP** einen relativ hohen Wert verwenden. Wenn zwei Programme versuchen, eine Kommunikationsverbindung einzurichten, benötigen beide für den Versuch eines **OPEN** etwa die halbe Zeit ihrer Ausführung.

Sämtliche Syntaxfehler in der Anweisung **OPEN COM** führen zu der Fehlermeldung `Unzulässiger Dateiname`.

Beispiel

Das folgende Programmfragment eröffnet den Datenübertragungskanal 1 im Direktzugriffsmodus mit einer Übertragungsrate von 9600 Baud, keinem Paritätsbit, 8 Datenbits und einem Stopp-Bit. Eingabe/Ausgabe erfolgt im Binärmodus. Andere Zeilen im Programm können jetzt auf Kanal 1 als Dateinummer 2 zugreifen.

```
OPEN "COM1:9600,N,8,1,BIN" AS #2
```

OPTION BASE-Anweisung

Funktion

Deklariert die untere Grenze für Datenfeldindizes.

Syntax

OPTION BASE *n*

Anmerkungen

Die Anweisung **OPTION BASE** ist niemals unbedingt erforderlich. Sie wird im allgemeinen dazu benutzt, die voreingestellte untere Grenze für Datenfelder zu verändern.

Der Wert von *n* muß entweder 0 oder 1 sein. Der Standardwert ist 0. Wenn die Anweisung

```
OPTION BASE 1
```

ausgeführt wird, ist 1 der niedrigste Wert, den ein Datenfeldindex haben kann.

Hinweis Die **TO**-Klausel in der **DIM**-Anweisung bietet eine einfachere und flexiblere Möglichkeit, den Bereich von Datenfeldindizes zu kontrollieren.

Wenn die untere Grenze eines Datenfeldindexes nicht explizit festgelegt wird, kann **OPTION BASE** benutzt werden, um die untere voreingestellte Grenze auf 1 zu verändern.

Die Anweisung **OPTION BASE** kann nur einmal in einem Modul (Quelldatei) benutzt werden und kann nur im Modul-Ebenen-Code erscheinen.

Eine **OPTION BASE**-Anweisung muß verwendet werden, bevor irgendwelche Datenfelder dimensioniert werden.

Verkettete Programme können eine **OPTION BASE**-Anweisung enthalten, wenn zwischen ihnen keine Datenfelder in **COMMON** übergeben werden oder die festgelegte Basis in den verketteten Programmen identisch ist. Das zu verkettende Programm übernimmt den Wert von **OPTION BASE** aus dem verkettenden Programm, wenn **OPTION BASE** im letzteren nicht angegeben ist.

Vergleichen Sie auch

DIM, LBOUND, REDIM

Beispiel

Die nachstehende Anweisung hebt den Standardwert Null auf, so daß der niedrigste Wert, den ein Datenfeldindex in diesem Programm haben kann, 1 ist:

```
OPTION BASE 1
DIM A(20)
PRINT LBOUND(A), UBOUND(A)
```

Ausgabe

```
1            20
```

OUT-Anweisung

Funktion

Sendet ein Byte zu einem Maschinen-E/A-Anschluß.

Syntax

OUT *Anschluß, Daten*

Anmerkungen

Die folgende Liste beschreibt die Argumente der **OUT**-Anweisung:

Argument	*Beschreibung*
Anschluß	Die Nummer des Anschlusses (Port). Die Nummer muß ein ganzzahliger Ausdruck im Bereich von 0 bis 65.535 sein.
Daten	Die Daten, die an den Anschluß gesendet werden. Es muß ein ganzzahliger Ausdruck im Bereich von 0 bis 255 sein.

Die Anweisungen **OUT** und **INP** geben einem BASIC-Programm direkte Kontrolle über die Hardware eines Systems durch die E/A-Anschlüsse. Diese Anweisungen sind vorsichtig zu verwenden, da sie die Hardware direkt manipulieren.

Vergleichen Sie auch

INP, WAIT

Beispiel

Das folgende Beispiel benutzt **OUT** und **INP**, um den Timer und den Lautsprecher zu steuern und einen Laut zu erzeugen.

Hinweis Dieses Beispiel ist spezifisch für den IBM-PC und 100%-Kompatible; führt auf anderen Rechnern jedoch u. U. zu unvorhersehbaren Ergebnissen.

```
' Spiele eine Tonleiter unter Benutzung des
' Lautsprechers und des Timers.
CONST WHOLE=5000!, QRTR=WHOLE/4.
CONST C=523.0, D=587.33, E=659.26, F=698.46, G=783.99
CONST A=880.00, B=987.77, C1=1046.50
CALL Sounds(C,QRTR) : CALL Sounds(D,QRTR)
CALL Sounds(E,QRTR) : CALL Sounds(F,QRTR)
CALL Sounds(G,QRTR) : CALL Sounds(A,QRTR)
CALL Sounds(B,QRTR) : CALL Sounds(C1,WHOLE)
' Benutze Anschluß 66, 67 und 97, um den Timer und den
' Lautsprecher zu kontrollieren und einen Laut zu
' erzeugen.
SUB Sounds (Freq!,Laenge!) STATIC
' Dividiere die Uhrenfrequenz durch die Tonfrequenz,
' um die Anzahl der "Ticks" zu berechnen, die die Uhr
' erzeugen muß.
   Ticks%=CINT(1193280!/Freq!)
   LoByte%=Ticks% AND &H00FF
   HiByte%=Ticks%\256
' Teile Timer mit, daß Daten kommen
   OUT 67,182
' Sende das niederwertige Byte, gefolgt vom
' höherwertigen Byte des Ergebnisses.
   OUT 66,LoByte%
   OUT 66,HiByte%
' Schalte den Lautsprecher ein, indem die Bits 0 und 1
' des PPI-Chips gesetzt werden.
```

```
' Nehme den aktuellen Wert und schalte die Bits ein.
   LautEin%=INP(97) OR &H03
   OUT 97,LautEin%
' Lasse den Lautsprecher für eine kurze Zeit
' eingeschaltet.
   FOR I! = 1 TO Laenge! : NEXT I!
' Schalte den Lautsprecher aus.
   LautAus%=INP(97) AND &HFC
   OUT 97, LautAus%
END SUB
```

PAINT-Anweisung

Funktion

Füllt eine Grafikfläche mit der angegebenen Farbe oder dem angegebenen Muster aus.

Syntax

PAINT [**STEP**](*x,y*)[,[*Ausfüllen*][,[*Randfarbe*][, *Hintergrund*]]]

Anmerkungen

Die folgende Liste beschreibt die Teile der **PAINT**-Anweisung:

Teil	*Beschreibung*
STEP	Definiert die Koordinaten relativ zu dem letzten gezeichneten Punkt. Wenn zum Beispiel der letzte gezeichnete Punkt (10,10) war, sind die Koordinaten, auf die **STEP** (4,5) Bezug nimmt, (4+10,5+10) oder (14,15).
(*x,y*)	Die Koordinaten, an denen das Ausfüllen beginnt. Dieser Punkt muß innerhalb oder außerhalb der Figur liegen, die ausgefüllt werden soll, nicht auf deren Rand. Wenn sich dieser Punkt innerhalb der Fläche befindet, wird das Innere der Figur ausgefüllt; befindet er sich außerhalb, so wird der Hintergrund ausgefüllt.

Teil	*Beschreibung*
Ausfüllen	Ein numerischer oder Zeichenkettenausdruck. Wenn *Ausfüllen* ein numerischer Ausdruck ist, muß die Nummer eine zulässige Farbe sein. Diese Farbe dient zum Ausfüllen der Fläche. Falls Sie *Ausfüllen* nicht angeben, wird die Textfarbe benutzt. (Informationen über zulässige Farben, Farbnummern und Farbattribute finden Sie unter **COLOR**, **PALETTE** und **SCREEN**.) Wenn das Argument *Ausfüllen* ein Zeichenkettenausdruck ist, füllt **PAINT** die Figur mit einem Muster, und nicht mit einer einzigen Farbe aus. Dies ist ähnlich der Geradenstruktur, mit der gestrichelte Geraden und keine durchgehenden Geraden gezeichnet werden. Siehe unten, "Ausfüllen mit Mustern".
Randfarbe	Ein numerischer Ausdruck, der das zu verwendende Farbattribut für den Rand der Figur angibt. Wird die Randfarbe erreicht, stoppt das Ausfüllen der aktuellen Bildzeile. Wird *Randfarbe* nicht angegeben, so wird das Argument *Ausfüllen* verwendet.
Hintergrund	Ein Zeichenkettenwert, der den "Hintergrund-Musterausschnitt" angibt, der beim Prüfen auf Abschluß des Randes zu überspringen ist. Der Ausfüllvorgang wird beendet, wenn nebeneinanderliegende Punkte die Ausfüllfarbe zeigen. Die Angabe eines Hintergrund-Musterausschnitts ermöglicht es Ihnen, über einen bereits ausgefüllten Bereich hinauszugehen. Wird *Hintergrund* nicht angegeben, so ist die Vorgabe CHR$(0).

Der Ausfüllvorgang ist beendet, wenn eine Bildzeile gezeichnet wird, ohne daß sich die Farbe irgendeines Bildpunktes ändert - mit anderen Worten, wenn die ganze Bildzeile gleich der Ausfüllfarbe ist.

Der Befehl **PAINT** erlaubt die Benutzung von Koordinaten außerhalb des Bildschirms oder des Darstellungsfeldes.

Ausfüllen mit Mustern

Als Ausfüllen mit Mustern wird die Verwendung eines **PAINT**-Musters bezeichnet, das pro Ebene 8 Bits breit und bis zu 64 Bytes lang ist. In der Musterzeichenkette maskiert jedes Byte bei der Abbildung von Punkten 8 Bits entlang der *x*-Achse. Die Syntax zur Erstellung dieser Maske lautet:

PAINT *(x,y)*,**CHR$** *(Arg1)*+**CHR$** *(Arg2)*+...+**CHR$** *(Argn)*

Dabei sind die Argumente zu **CHR$** Zahlen zwischen 0 und 255, die in Binär-Form über die *x*-Achse des Musters dargestellt werden. Bis zu 64 dieser **CHR$**-Elemente sind möglich; jedes erzeugt ein Bild der Bit-Anordnung des Codes für dieses Zeichen und nicht des zugewiesenen Zeichens selbst. Beispielsweise ist die Dezimalzahl 85 binär "01010101"; die von CHR$(85) auf einem Schwarzweiß-Bildschirm erzeugte Grafikbildgerade ist eine Gerade aus 8 Bildpunkten, bei der die geradzahligen Punkte weiß und die ungeradzahligen Punkte schwarz sind. Das heißt, in einem Schwarzweiß-System schaltet jedes Bit gleich 1 den zugehörigen Bildpunkt ein und jedes Bit gleich 0 den zugehörigen Bildpunkt aus. Das ASCII-Zeichen CHR$(85), das für U steht, wird in diesem Fall nicht angezeigt.

Wenn das Argument *Hintergrund* angegeben wird, legt es den "Hintergrund-Musterausschnitt" fest, der beim Prüfen auf Abschluß des Randes zu überspringen ist. Sie können maximal zwei aufeinanderfolgende Bytes angeben, die der Musterzeichenkette im Hintergrund-Musterausschnitt entsprechen. Die Angabe von mehr als zwei Bytes führt zur Fehlermeldung `Unzulässiger Funktionsaufruf.`

Das Ausfüllen mit Mustern kann ebenso verschiedene Muster in verschiedenen Farben darstellen. In Kapitel 5, "Grafiken", in *Programmieren in BASIC: Ausgewählte Themen* finden Sie eine detaillierte Beschreibung des Ausfüllens mit Mustern.

Vergleichen Sie auch

CHR$, CIRCLE, DRAW, LINE, SCREEN

Beispiel

Das folgende Programm zeichnet einen violetten Fisch mit kobaltblauer Schwanzflosse:

```
CONST PI=3.1415926536
CLS
SCREEN 1
CIRCLE (190,100),100,1,,,.3  'Umrisse des Fisches in
                             'kobaltblau.
CIRCLE (265,92),5,1,,,.7     'Umrisse des Fischauges
                             'in kobaltblau.
PAINT (190,100),2,1          'Körper des Fisches in
                             'violett ausfüllen.
LINE (40,120)-STEP (0,-40),2 'Umrisse der
LINE -STEP (60,+20),2        'Schwanzflosse
LINE -STEP (-60,+20),2       'in violett.
PAINT (50,100),1,2       'Füllt Schwanzflosse
                         'kobaltblau aus.
```

```
CIRCLE (250,100),30,0,PI*3/4,PI* 5/4,1.5 'Zeichnet
                                    'Kiemen in schwarz.
FOR Y = 90 TO 110 STEP 4
   LINE (40,Y)-(52,Y),0     'Zeichnet Muster in
NEXT                        'Schwanzflosse.
```

PALETTE-, PALETTE USING-Anweisungen

Funktion

Ändert eine oder mehrere Farben in der Palette.

Syntax

PALETTE [*Attribut, Farbe*]
PALETTE USING *Datenfeldname* [(*Datenfeldindex*)]

Anmerkungen

Die Anweisung **PALETTE** hat folgende Argumente:

Argument	***Beschreibung***
Attribut	Das Paletten-Attribut, das geändert werden soll.
Farbe	Die Nummer der Bildschirmfarbe, die dem Attribut zugewiesen werden soll. Die *Farbe* muß ein langganzzahliger Ausdruck für die IBM Video-Grafikkarte (Video Graphics Array Adapter, VGA) und die Mehrfarben-Grafikkarte (Multicolor Graphics Array, MCGA) im Bildschirmmodus 11 bis 13 sein. Ganzzahlige oder langganzzahlige Ausdrücke können mit dem Erweiterten Grafikadapter (Enhanced Graphics Adapter, EGA) verwendet werden.
Datenfeldname	Ein Datenfeld, das die Nummern der Farben enthält, die den im aktuellen Bildschirmmodus zur Verfügung stehenden Attributen zugewiesen werden. Die Adapter VGA und MCGA erfordern ein langganzzahliges Datenfeld im Bildschirmmodus 11 bis 13. Mit der EGA-Karte kann dies entweder ein ganzzahliges oder ein langganzzahliges Datenfeld sein.
Datenfeldindex	Der Index des ersten Datenfeldelementes, der benutzt wird, um die Palette zu setzen.

Die Anweisung **PALETTE** arbeitet *nur* mit Systemen, die mit EGA-, VGA- oder MCGA-Adaptern ausgerüstet sind.

Die Anweisung bietet eine Möglichkeit, Bildschirmfarben (die eigentlichen binären Werte, die vom Adapter benutzt werden) Farbattributen (ein kleiner Satz von Werten) zuzuordnen. Alle BASIC-Grafikanweisungen wie **CIRCLE, COLOR, DRAW** oder **LINE** benutzen Farbattribute und nicht Werte der Bildschirmfarben.

Wenn ein Programm in einen Bildschirmmodus geht, werden die Attribute auf eine Reihe voreingestellter Werte gesetzt. (Unter der Beschreibung zu der **SCREEN**-Anweisung finden Sie eine Liste der voreingestellten Farben.) Für die Modi EGA, VGA und MCGA wurden die Werte so ausgewählt, daß der Bildschirm die gleichen Farben zeigt, obwohl die EGA-Karte andere Farbwerte benutzt.

Mit der **PALETTE**-Anweisung können Sie den Attributen verschiedene Farben zuordnen. Hierdurch erhalten Sie eine bessere Kontrolle über die Farben auf dem Bildschirm.

Eine **PALETTE**-Anweisung ohne Argumente stellt die Palette zurück auf die voreingestellten Werte.

Wenn Sie eine **PALETTE**-Anweisung mit Argumenten ausführen, benutzt der Adapter die Bildschirmfarbe (angezeigt durch *Farbe*) jedesmal, wenn der Wert *Attribut* in einer Anweisung wie **DRAW** oder **LINE**, die eine Farbe bezeichnet, auftritt. Mit dem Verändern der Bildschirmfarbe, die einem Attribut zugeordnet ist, verändert sich die Farbe auf dem Bildschirm sofort.

Nehmen Sie beispielsweise an, daß die aktuelle Palette die Farben 0, 1, 2 und 3 in den vier numerierten Attributen 0, 1, 2 und 3 enthält. Die folgende **DRAW**-Anweisung

```
DRAW "C3L100"
```

wählt Attribut 3 und zeichnet eine Linie aus 100 Bildpunkten, wofür sie die mit Attribut 3 verbundene Farbe benutzt, in diesem Fall ebenfalls 3. Bei der Ausführung der Anweisung

```
PALETTE 3,2
```

wird die mit Attribut 3 verbundene Farbe in Farbe 2 geändert. Der gesamte Text oder alle Grafiken, die im Augenblick unter Verwendung von Attribut 3 auf dem Bildschirm angezeigt werden, werden sofort in Farbe 2 geändert. Texte oder Grafiken, die anschließend mit Attribut 3 angezeigt werden, werden ebenfalls in Farbe 2 angezeigt. Die neue Palette von *Farben* enthält 0, 1, 2 und 2.

Mit der Option **USING** können alle Einträge in der Palette mit einer **PALETTE**-Anweisung geändert werden. Das Argument *Datenfeldname* ist der Name eines ganzzahligen oder langganzzahligen Datenfeldes, und *Datenfeldindex* gibt den Index des ersten Datenfeldelementes in *Datenfeldname* an, das beim Setzen der Palette zu verwenden ist. Jedem *Attribut* in der Palette wird eine entsprechende *Farbe* aus diesem Datenfeld zugewiesen. Das Datenfeld muß groß genug dimensioniert werden, um alle Paletteneinträge nach *Datenfeldindex* setzen zu können. Wenn Sie beispielsweise allen 16 Attributen Farben zuweisen und der Index des ersten Datenfeldelementes in Ihrer Anweisung **PALETTE USING** 5 ist, muß das Datenfeld dimensioniert werden, um mindestens 20 Elemente aufnehmen zu können, weil die Anzahl der Elemente von 5 bis einschließlich 20 16 ist:

```
DIM PAL%(20)
.
.
.
PALETTE USING PAL%(5)
```

Ein *Farbe*-Argument von -1 läßt das Attribut ungeändert. Alle anderen negativen Zahlen sind unzulässige Werte für *Farbe*.

Sie können die Anweisung **COLOR** benutzen, um die Attribute der voreingestellten Textfarbe und Hintergrundfarbe des Bildschirms zu setzen. Das Argument der Textfarbe gibt die Art und Weise an, wie Textzeichen auf dem Bildschirm erscheinen. Bei einer normalen Anfangs-Paletteneinstellung erscheinen mit Attribut 0 dargestellte Punkte auf dem Bildschirm schwarz. Mit der Anweisung **PALETTE** können Sie zum Beispiel die Abbildung von Attribut 0 von schwarz in weiß ändern.

Die Tabelle weiter unten führt die *Attribut*- und *Farben*-Bereiche für verschiedene Adaptertypen und Bildschirmmodi auf.

Die VGA verwendet einen anderen Weg als die EGA, um Farbwerte zu berechnen. Wählen Sie die Intensität von rot, grün und blau, um einen Farbwert zu berechnen. Die Intensität einer Farbe ist eine Zahl von 0 (niedrige Intensität) bis 63 (hohe Intensität). Benutzen Sie dann die folgende Formel, um die Farbnummer zu berechnen:

Farbnummer = **65536 *** *blau* + **256 *** *grün* + *rot*

Weil es im Bereich der Farbnummern Zwischenräume gibt, sollten Sie die Formel verwenden, anstatt einfach eine Zahl anzugeben.

Bei Verwendung des IBM Analog-Monochrom-Monitors werden die VGA-Farbwerte in einen Grauskala-Wert übertragen, indem eine gewichtete Summe der Intensitäten von rot, blau und grün verwendet wird:

Grauwert = **11%** *blau* + **59%** *grün* + **30%** *rot*

Wenn zum Beispiel die Intensitäten von blau, grün und rot 45, 20 und 20 sind, wäre der Grauwert 0,11*45+0,59*20+0,30*20 oder 22 (die Nachkommastellen des Ergebnisses werden abgeschnitten).

Die Liste der verfügbaren Farben für verschiedene Kombinationen von **SCREEN**-Modi, Monitoren und Grafikadaptern finden Sie im Nachschlageteil unter der Anweisung **SCREEN**.

SCREEN-Modus	***Angeschlossener Monitor***		***Adapter-bereich***	***Attribut-Farb-bereich***
0	Monochrom	MDPA	0-15	NA
	Monochrom	EGA	0-15	0-2
	Farbe	CGA	0-15	NA
	Farbe/ Erweitert[c]	EGA	0-15	0-63
	NA	VGA	0-15	0-63
	NA	MCGA	0-15	NA
1	Farbe	CGA	0-3	NA
	Farbe/ Erweitert[c]	EGA	0-3	0-15
	NA	VGA	0-3	0-15
	NA	MCGA	0-3	NA
2	Farbe	CGA	0-1	NA
	Farbe/ Erweitert[c]	EGA	0-1	0-15
	NA	VGA	0-1	0-15
	NA	MCGA	0-1	NA
7	Farbe/ Erweitert[c]	EGA	0-15	0-15
	NA	VGA	0-15	0-15
8	Farbe/ Erweitert[c]	EGA	0-15	0-15
	NA	VGA	0-15	0-15
9	Erweitert[c]	EGA[a]	0-3	0-63
	Erweitert[c]	EGA[b]	0-15	0-63
	NA	VGA	0-16	0-63

SCREEN-Modus	*Angeschlossener Monitor*		*Adapter-bereich*	*Attribut-Farb-bereich*
10	Monochrom	EGA	0-3	0-8
	NA	VGA	0-3	0-8
11	NA	VGA	0-1	0-262.143[d]
	NA	MCGA	0-1	0-262.143[d]
12	NA	VGA	0-15	0-262.143[d]
13	NA	VGA	0-255	0-262.143[d]
	NA	MCGA	0-255	0-262.143[d]

a NA = nicht anwendbar
b Mit 64K EGA-Speicher
c Mehr als 64K EGA-Speicher
d Erweiterter IBM-Farbbildschirm
e Bildschirmfarben werden nicht von 0 bis 262.143 numeriert. Sehen Sie in der Beschreibung zu der Anweisung COLOR nach.

Hinweis Wegen ihres großen Bereichs von Farben benötigen die VGA- und MCGA-Karte langganzzahlige Datenfelder in der **PALETTE USING**-Anweisung in den Bildschirmmodi 11 - 13.

Vergleichen Sie auch

CIRCLE, COLOR, DRAW, LINE, SCREEN

Beispiele

Die folgenden Zeilen zeigen die verschiedenen Formen der **PALETTE**- und **PALETTE USING**-Anweisungen:

```
PALETTE 0,2       'Ändert alle farbigen Punkte mit
                  'Attribut 0 in Farbe 2
PALETTE USING A%(0)  'Ändert jeden Paletteneintrag. Da
                     'das Datenfeld auf Null
                     'initialisiert wurde, werden
                     'jetzt alle Attribute auf
                     'Anzeigefarbe Null gesetzt. Der
                     'Bildschirm erscheint in einer
                     'einzigen Farbe. Die Ausführung
                     'von BASIC-Anweisungen ist jedoch
                     'weiterhin möglich.
```

```
PALETTE       'Setzt jeden Paletteneintrag auf seine
              'entsprechende Anfangs-Anzeigefarbe. Die
              'tatsächlichen Anfangsfarben Ihres
              'Bildschirms hängen von der benutzten
              'Hardware ab.
```

PCOPY-Anweisung

Funktion

Kopiert den Inhalt einer Bildschirmseite in eine andere.

Syntax

PCOPY *Quellseite, Zielseite*

Anmerkungen

Quellseite ist ein ganzzahliger Ausdruck im Bereich von 0 bis *n*, wobei *n* die maximale Anzahl von Seiten ist; diese wird durch die aktuelle Bildspeichergröße und die Seitengröße für den aktuellen Bildschirmmodus festgelegt.

Zielseite stellt dieselben Anforderungen wie *Quellseite.*

Unter der Beschreibung zur **SCREEN**-Anweisung finden Sie weitere Informationen über die Anzahl von Seiten, die in verschiedenen Modi verfügbar sind.

Vergleichen Sie auch

CLEAR, SCREEN

Beispiel

Dieses Beispiel kopiert den Inhalt von Seite 1 auf Seite 2:

```
PCOPY 1,2
```

PEEK-Funktion

Funktion

Gibt das in der angegebenen Speicheradresse gespeicherte Byte an.

Syntax

PEEK(*Adresse*)

Anmerkungen

Der ermittelte Wert ist eine ganze Zahl im Bereich von 0 bis 255. Das Argument *Adresse* ist ein Wert im Bereich von 0 bis 65.535. Das Argument *Adresse* wird als Offset des aktuellen Segmentes behandelt (wie durch die **DEF SEG**-Anweisung eingestellt).

Wenn das Argument ein Gleitkommawert einfacher oder doppelter Genauigkeit oder eine lange Ganzzahl ist, wird es in eine 2-Byte-Ganzzahl umgewandelt.

Die Funktion **PEEK** ergänzt die Anweisung **POKE**.

Vergleichen Sie auch

DEF SEG, POKE, VARPTR

Beispiel

Siehe Beispiel für **DEF SEG.**

PEN-Funktion

Funktion

Liest die Lichtstift (Lightpen)-Koordinaten ein.

Syntax

PEN(*n*)

Anmerkungen

Das Argument *n* zeigt an, welcher Wert ausgegeben werden soll. Es ist ein numerischer Ausdruck im Bereich von 0 bis 9.

Hinweis Die **PEN**-Funktion ist nicht aktiv, wenn der Maustreiber eingeschaltet ist, da der Maustreiber die BIOS-Aufrufe der **PEN**-Funktion benutzt. Benutzen Sie die Mausfunktion 14, um die Lichtstift-Emulation des Treibers auszuschalten. Die Mausfunktion 13 schaltet die Emulation wieder ein. In Ihrem Maushandbuch werden Sie weitere Informationen finden.

Die folgende Liste beschreibt die Werte für *n* und die entsprechenden Werte, die dazu von **PEN** ausgegeben werden:

Argument	*Zurückgegebener Wert*
0	Den letzten Lichtstiftstatus: -1, wenn der Lichtstift seit der letzten Anfrage unten war (pen down), sonst 0.
1	Die x-Koordinate des Bildpunktes, an dem der Lichtstift zuletzt aktiviert wurde.
2	Die y-Koordinate des Bildpunktes, an dem der Lichtstift zuletzt aktiviert wurde.
3	Der aktuelle Schalterwert des Lichtstiftes: -1, wenn er unten ist; 0, wenn er oben ist.
4	Die letzte bekannte gültige x-Koordinate des Bildpunktes.
5	Die letzte bekannte gültige y-Koordinate des Bildpunktes.
6	Die Zeilenposition eines Zeichens, in welcher der Lichtstift zuletzt aktiviert wurde.
7	Die Spaltenposition eines Zeichens, in welcher der Lichtstift zuletzt aktiviert wurde.
8	Die letzte bekannte Zeichenzeile, in welcher der Lichtstift positioniert war.
9	Die letzte bekannte Zeichenspalte, in welcher der Lichtstift positioniert war.

Beispiel

Das nachstehende Beispiel erzeugt eine Endlosschleife zur Ausgabe des aktuellen Status (OBEN/UNTEN):

```
CLS
PEN ON
DO
   P = PEN(3)
   LOCATE 1,1 : PRINT "Lichtstift ist ";
   IF P THEN PRINT "UNTEN" ELSE PRINT "OBEN"
   X = PEN(4) : Y = PEN(5)
   PRINT "X =" X, "Y =" Y
LOOP
```

PEN ON-, OFF- und STOP-Anweisungen

Funktion

Aktiviert, deaktiviert oder unterbricht die Ereignisverfolgung für den Lichtstift (Lightpen).

Syntax

PEN ON

PEN OFF

PEN STOP

Anmerkungen

Die **PEN ON**-Anweisung schaltet die Lichtstift-Ereignisverfolgung mit einer **ON PEN**-Anweisung ein. Der Stift ist zunächst ausgeschaltet. Ein Lichtstift-Ereignis tritt ein, wenn der Lichtstift durch das Drücken der Spitze des Stiftes auf den Bildschirm oder durch Drücken des Tastringes aktiviert wird. Eine **PEN ON**-Anweisung muß vor irgendeinem Aufruf einer Stiftlesefunktion ausgeführt werden. Wenn eine Stiftlesefunktion aufgerufen wird, wenn der Stift ausgeschaltet ist, erfolgt die Fehlermeldung `Unzulässiger Funktionsaufruf`.

Die Anweisung **PEN OFF** deaktiviert die Ereignisverfolgung für den Lichtstift.

Die Anweisung **PEN STOP** unterbricht die Ereignisverfolgung für den Lichtstift: Ein **PEN**-Ereignis wird vermerkt und verfolgt, sobald die Ereignisverfolgung aktiviert wird.

Zur Beschleunigung der Programmausführung sollte der Lichtstift mit einer **PEN OFF**-Anweisung ausgeschaltet werden, sofern diese nicht benötigt wird.

Hinweis Der Lichtstift erfordert einen IBM Farbgrafikadapter.

Vergleichen Sie auch

ON *Ereignis*, **PEN-Funktion**

PLAY-Anweisung

Funktion

Spielt die Musik, die durch eine Zeichenkette angegeben wird.

Syntax

PLAY *Befehls-Zeichenkette*

Anmerkungen

Befehls-Zeichenkette ist ein Zeichenkettenausdruck, der einen oder mehrere der unten aufgelisteten Befehle enthält.

Die Anweisung **PLAY** verwendet ein ähnliches Konzept wie **DRAW**: Eine Musik-Makrosprache wird (wie nachstehend beschrieben) in einer Anweisung zusammengefaßt. Eine Gruppe von Befehlen, die als Teil der Anweisung **PLAY** verwendet wird, legt eine bestimmte Aktion fest.

In kompilierten Programmen sollten Sie für Variablen die Form **VARPTR$**(*Variable*) verwenden. Beispielsweise sollten die folgenden BASICA-Anweisungen

```
PLAY "XA$"
PLAY "O=I"
```

für den Compiler folgendermaßen geschrieben werden:

```
PLAY "X" + VARPTR$(A$)
PLAY "O=" + VARPTR$(I)
```

Die Musikmakros von *Befehls-Zeichenkette* werden nachstehend erläutert:

Oktave	*Aktion*
>	Erhöht die Oktave um 1. Oktave wird nicht größer als 6, bleibt aber bei 6.
<	Vermindert die Oktave um 1. Oktave wird nicht kleiner als 0, bleibt aber bei 0.
o *n*	Setzt die laufende Oktave. Es gibt 7 Oktaven, die von 0 bis 6 numeriert sind.

Ton	*Aktion*
A-G	Spielt einen Ton im Bereich **A** bis **G**. Ein "#"- oder ein "+"-Symbol hinter einer Note zeigt einen Halbton höher, ein "-" einen Halbton tiefer an.
N *n*	Spielt den Ton *n*. *n* kann im Bereich von 0 bis 84 liegen (die sieben möglichen Oktaven enthalten 84 Töne); *n* = 0 bedeutet eine Pause.

Länge	*Aktion*
L *n*	Setzt die Länge jedes Tons. **L** 4 ist eine Viertelnote, **L** 1 eine ganze Note usw. *n* kann im Bereich von 1 bis 64 liegen. Die Längenangabe kann auch hinter einer Note stehen, wenn nur die Länge dieser Note geändert werden soll. In einem solchen Fall ist z. B. **A** 16 gleichbedeutend mit **L** 16 **A**.
MN	Setzt "Musik normal", so daß jeder Ton 7/8 der durch die Länge (**L**) festgelegten Zeit gehalten wird.
ML	Setzt "Musik legato", so daß jeder Ton die durch die Länge (**L**) gesetzte volle Zeit gehalten wird.
MS	Setzt "Musik stakkato", so daß jeder Ton 3/4 der durch die Länge (**L**) festgelegten Zeit gehalten wird.

Tempo	*Aktion*
P *n*	Gibt eine Pause im Bereich von 1 bis 64 an. Die Option entspricht der Länge jedes Tons, die mit **L** *n* gesetzt wird.
T *n*	Setzt das "Tempo" oder die Anzahl der Viertelnoten (**L** 4) in einer Minute. *n* kann im Bereich von 32 bis 255 liegen. Die Vorgabe für *n* ist 120.

Anwendung	*Aktion*
MF	Setzt Töne (**PLAY**-Anweisung) und Klang (**SOUND**) so, daß sie im Vordergrund laufen. Dies bedeutet, daß jeder nachfolgende Ton und jeder nachfolgende Klang erst dann beginnen kann, wenn der vorhergehende Ton oder der vorhergehende Klang beendet ist. Dies ist die Standardeinstellung.
MB	Setzt Töne (**PLAY**-Anweisung) und Klang (**SOUND**) so, daß sie im Hintergrund laufen. Dies bedeutet, daß jeder Ton oder jeder Klang in einem Puffer zwischengespeichert wird. Auf diese Weise kann das BASIC-Programm weiter ausgeführt werden, während der Ton oder der Klang im Hintergrund hörbar ist. Die maximale Anzahl der Töne, die gleichzeitig im Hintergrund gespielt werden können, ist 32.

Zusätze	*Aktion*
# oder +	Steht hinter einer bestimmten Note und erhöht den Ton um einen Halbton.
-	Steht hinter einer bestimmten Note und vermindert den Ton um einen Halbton.
.	Ein Punkt hinter einer Note bewirkt, daß der Ton 3/2 mal so lang erklingt, wie durch **L** (Länge) mal **T** (Tempo) festgelegt wird. Der Punkt hat dieselbe Bedeutung wie in der Musik. Hinter einer Note können mehrere Punkte stehen. Jeder Punkt addiert eine Länge gleich 1/2 der Länge des vorhergehenden Punktes hinzu. A. spielt 1 + 1/2, oder 3/2 mal die Länge; A.. spielt 1 + 1/2 + 1/4, oder 7/4 mal die Länge usw. Punkte können auch nach einer Pause (**P**) stehen. In diesem Fall kann die Länge der Pause genauso wie die Länge der Töne berechnet werden.

Teilfolge	*Aktion*
"X"+**VARPTR$**(*Zeichenkette*)	Führt eine Teilfolge aus.

Wegen der langsam getakteten Unterbrechungsgeschwindigkeit werden einige Töne nicht mit höheren Tempi gespielt (z. B. **L** 64 bei **T** 255).

Beispiele

Das nachstehende Beispiel benutzt ">", um die Tonleiter aufwärts von Oktave 0 bis Oktave 6 zu spielen, und spielt sie anschließend mit "<" von Oktave 6 bis Oktave 0 abwärts.

```
TONL$ = "CDEFGAB"
PLAY "o0 X" + VARPTR$ (TONL$)
FOR I = 1 TO 6
   PLAY ">X" + VARPTR$(TONL$)
NEXT
```

```
PLAY "o6 X" + VARPTR$ (TONL$)
FOR I = 1 TO 6
   PLAY "<X" + VARPTR$(TONL$)
NEXT
```

Das folgende Beispiel spielt den Anfang des ersten Satzes von Beethovens Fünfter Symphonie:

```
THEMA$ = "T180 o2 P2 P8 L8 GGG L2 E-"
WIEDERHOL$ = "P24 P8 L8 FFF L2 D"
PLAY THEMA$ + WIEDERHOL$
```

PLAY-Funktion

Funktion

Gibt die Anzahl der Noten an, die augenblicklich im Puffer für die Hintergrundmusik stehen.

Syntax

PLAY(*n*)

Anmerkungen

Das Argument *n* ist ein Hilfsargument und kann jeder beliebige Wert sein.

PLAY(*n*) gibt 0 an, wenn der Musik-Vordergrundmodus aktiv ist.

Vergleichen Sie auch

ON *Ereignis*, **PLAY-Anweisung, PLAY ON, OFF** und **STOP**

Beispiel

Siehe Beispiele auf den Nachschlageseiten zu **ON** *Ereignis*.

PLAY ON-, OFF- und STOP-Anweisungen

Funktion

PLAY ON aktiviert die Ereignisverfolgung für die Musik, **PLAY OFF** deaktiviert die Ereignisverfolgung für die Musik und **PLAY STOP** unterbricht die Ereignisverfolgung für die Musik.

Syntax

PLAY ON

PLAY OFF

PLAY STOP

Anmerkungen

Diese Anweisungen werden mit der **ON PLAY**-Anweisung benutzt, um Musik-Ereignisse zu verfolgen.

Wenn eine **PLAY OFF**-Anweisung ausgeführt wird, wird die Unterroutine der Ereignisverfolgung nicht ausgeführt und das Ereignis wird nicht vermerkt.

Eine **PLAY STOP**-Anweisung führt keine Ereignisverfolgungs-Unterroutine aus, die Unterroutine wird jedoch ausgeführt, sobald eine **PLAY ON**-Anweisung ausgeführt wurde.

Wenn eine Ereignisverfolgung erfolgt (d. h., das **GOSUB** wird ausgeführt), wird **PLAY STOP** automatisch ausgeführt, so daß rekursive Verfolgungen nicht vorkommen können. Die Rückkehr (**RETURN**) aus der Verfolgungsroutine führt automatisch eine **PLAY ON**-Anweisung aus, außer wenn **PLAY OFF** in der Unterroutine explizit ausgeführt wurde.

Beispiel

Ein Beispiel für die Ereignisverfolgung für Musik finden Sie unter der Beschreibung der Anweisung **ON** *Ereignis*.

PMAP-Funktion

Funktion

Bildet logische Koordinaten-Ausdrücke auf physikalische Orte oder physikalische Ausdrücke auf einen Ort logischer Koordinaten ab.

Syntax

PMAP (*Ausdruck, Funktion*)

Anmerkungen

Das Argument *Ausdruck* gibt die Koordinaten des abzubildenden Punktes an.

Das Argument *Funktion* kann einen der vier folgenden Werte haben:

Wert	***Beschreibung***
0	Bildet den logischen Koordinaten-Ausdruck auf der physikalischen x-Koordinate ab.
1	Bildet den logischen Koordinaten-Ausdruck auf der physikalischen y-Koordinate ab.
2	Bildet den physikalischen Ausdruck auf der logischen x-Koordinate ab.
3	Bildet den physikalischen Ausdruck auf der logischen y-Koordinate ab.

Die vier **PMAP**-Funktionen ermöglichen es dem Benutzer, gleichwertige Punktorte zwischen den mit der Anweisung **WINDOW** erstellten logischen Koordinaten und dem in der Anweisung **VIEW** definierten physikalischen Koordinatensystem des Bildschirms oder des Darstellungsfeldes herauszufinden.

Vergleichen Sie auch

VIEW, WINDOW

Beispiel

Das folgende Programmfragment benutzt **PMAP**, um Koordinatenwerte von logischen Koordinaten in Bildschirmkoordinaten und von Bildschirmkoordinaten in logische Koordinaten zu konvertieren:

```
SCREEN 2
'Die Koordinaten der oberen linken Ecke des in der
'folgenden Anweisung definierten Fensters sind
'(80,100); die Koordinaten der unteren rechten Ecke
'sind (200,200):
WINDOW SCREEN (80,100) - (200,200)
'Wenn die physikalischen Bildschirmkoordinaten der
'oberen linken Ecke (0,0) und der unteren rechten Ecke
'(639,199) sind, geben die folgenden Anweisungen
'Bildschirmkoordinaten an, die den logischen
'Koordinaten (80,100) entsprechen:
X = PMAP(80,0)     'X = 0
Y = PMAP(100,1)    'Y = 0
'Die folgenden Anweisungen geben die
'Bildschirmkoordinaten an, die den logischen
'Koordinaten (200,200) entsprechen:
X = PMAP(200,0)     'X = 639
Y = PMAP(200,1)     'Y = 199
'Die folgenden Anweisungen geben die logischen
'Koordinaten an, die den Bildschirmkoordinaten
'(639,199) entsprechen:
X = PMAP(639,2)     'X = 200
Y = PMAP(199,3)     'Y = 200
```

POINT-Funktion

Funktion

Liest die Farbnummer eines Bildpunktes vom Bildschirm oder gibt die Koordinaten des Bildpunktes an.

Syntax

POINT (*x,y*)

POINT (*Nummer*)

Anmerkungen

Die Koordinaten *x* und *y* identifizieren den durch **POINT** ausgewerteten Bildpunkt. Bei Aufruf mit einem Koordinatenpaar gibt **POINT** die Farbnummer des angegebenen Bildpunktes zurück. Wenn der angegebene Bildpunkt nicht im Bereich liegt, gibt **POINT** den Wert -1 an.

POINT mit einem Argument (siehe folgende Tabelle) ermöglicht dem Benutzer die Ermittlung der aktuellen Cursor-Koordinaten im Grafikmodus.

Argument	*Zurückgegebener Wert*
0	Die aktuelle physikalische *x*-Koordinate.
1	Die aktuelle physikalische *y*-Koordinate.
2	Die aktuelle logische *x*-Koordinate. Mit diesem Argument wird derselbe Wert wie bei der Funktion **POINT(0)** zurückgegeben, wenn die Anweisung **WINDOW** nicht verwendet wurde.
3	Die aktuelle logische *y*-Koordinate.Mit diesem Argument wird derselbe Wert wie bei der Funktion **POINT(1)** zurückgegeben, wenn die Anweisung **WINDOW** nicht verwendet wurde.

Beispiel

Das folgende Beispiel zeichnet die mit der Anweisung **CIRCLE** gezeichnete Ellipse neu; **POINT** wird benutzt, um den Rand der Ellipse zu finden, indem auf Farbänderung getestet wird:

```
DEFINT X,Y
PRINT "Neigungswinkel in Grad (0 bis 90) eingeben";
INPUT ":",WKL
SCREEN 1      'Mittlere Bildschirmauflösung
WKL = (3.1415926#/180)*WKL  'Wandelt Grad in
                             'Bogenmaß um
Cs = COS(WKL) : Sn = SIN(WKL)
CIRCLE (45,70),50,2,,,2  'Zeichnet Ellipse
PAINT (45,70),2      'Füllt Inneres der Ellipse aus
```

```
FOR Y = 20 TO 120
   FOR X = 20 TO 70
   'Prüft jeden Punkt im Rechteck, das die Ellipse
   'einschließt:
   IF POINT(X,Y) <> 0 THEN
      'Wenn der Punkt innerhalb der Ellipse liegt,
      'wird ein entsprechender Punkt in der geneigten
      'Ellipse dargestellt.
      Xneu = (X*Cs - Y*Sn) + 200 : Yneu = (X*Sn + _
      Y*Cs)
      PSET(Xneu,Yneu),2
   END IF
   NEXT
NEXT
END
```

POKE-Anweisung

Funktion

Schreibt ein Byte in eine Speicherstelle.

Syntax

POKE *Adresse, Byte*

Anmerkungen

Der Ausdruck *Adresse* ist ein Wert, der die Adresse der Speicherstelle darstellt; *Adresse* muß im Bereich von 0 bis 65.535 liegen.

Der Ausdruck *Byte* ist das zu schreibende Daten-Byte; er ist ein ganzzahliger Wert im Bereich von 0 bis 255.

Die *Adresse* wird als Offset des aktuellen Segmentes (wie durch die **DEF SEG**-Anweisung gesetzt) behandelt.

Wenn das Argument ein Gleitkommawert einfacher oder doppelter Genauigkeit ist oder eine lange Ganzzahl, wird er in eine 2-Byte-Ganzzahl umgewandelt.

Die ergänzende Funktion zu **POKE** ist **PEEK**.

Warnung Seien Sie vorsichtig bei der Verwendung von **POKE**. Wenn diese Anweisung falsch verwendet wird, kann sie Fehler in BASIC oder im Betriebssystem verursachen.

Vergleichen Sie auch

DEF SEG, PEEK, VARPTR

Beispiel

Siehe Beispiel für die Anweisung **DEF SEG.**

POS-Funktion

Funktion

Gibt die aktuelle horizontale Position des Cursors an.

Syntax

POS (*Spalte*)

Anmerkungen

Das Argument *Spalte* wird bei BASIC nicht benutzt. Die Position ganz links ist 1. Zur Ermittlung der aktuellen vertikalen Zeilenposition des Cursors verwenden Sie die Funktion **CSRLIN.**

Vergleichen Sie auch

CSRLIN, LPOS

Beispiel

Das folgende Beispiel benutzt **POS**, um die Eingabe auf einer neuen Zeile nach jeweils 40 Zeichen zu beginnen:

```
PRINT "Dieses Programm startet eine neue Zeile, ";
PRINT "nachdem jeweils vierzig Zeichen ausgegeben";
PRINT "wurden. Drücken Sie <STRG+C> zum Beenden."
PRINT
DO
   DO WHILE POS(0) < 41 'Gleiche Zeile, solange
                        ' weniger als 40 Zeichen
                        ' ausgegeben wurden
      DO
         Char$=INKEY$
      LOOP WHILE Char$=""
      'Wenn Eingabe gleich STRG+C, dann beenden;
      'andernfalls das Zeichen ausgeben:
      IF ASC(Char$) = 3 THEN END ELSE PRINT Char$;
   LOOP
   PRINT       'Gib eine neue Zeile aus
LOOP
```

PRESET-Anweisung

Funktion

Stellt einen angegebenen Punkt auf dem Bildschirm dar.

Syntax

PRESET [STEP] (*x-Koordinate, y-Koordinate*) [, *Farbe*]

Anmerkungen

PRESET funktioniert genauso wie **PSET**, außer daß bei Nichtangabe von *Farbe* die Hintergrundfarbe gewählt wird. Die folgende Liste beschreibt die Teile der Anweisung **PRESET**:

Teil	*Beschreibung*
STEP	Zeigt an, daß die gegebenen *x*- und *y*-Koordinaten nicht absolut sondern relativ sind. Die Koordinaten werden als Abstände von der letzten Cursorposition und nicht als Abstände von der Bildschirmkoordinate (0,0) behandelt. Wenn beispielsweise der letzte Punkt, auf den Bezug genommen wurde, (10,10) war, würde `PRESET STEP (10,5)` sich auf den Punkt (20,15) beziehen.
x-Koordinate	Die x-Koordinate des Bildpunktes, der gesetzt werden soll.
y-Koordinate	Die y-Koordinate des Bildpunktes, der gesetzt werden soll.
Farbe	Das Farbattribut für den angegebenen Bildpunkt.

Liegt eine Koordinate außerhalb des aktuellen Darstellungsfeldes, so erfolgt weder eine Aktion noch wird eine Fehlermeldung ausgegeben.

Vergleichen Sie auch

PSET

Beispiel

Das folgende Beispiel zeichnet eine zwanzig Bildpunkte lange Gerade. Die Gerade läuft von links nach rechts über den Bildschirm.

```
SCREEN 1 : COLOR 1,1 : CLS
FOR I = 0 TO 299 STEP 3
   FOR J = I TO 20+I
      PSET (J,50),2   'Zeichnet die Gerade an der
                      'neuen Position
      NEXT
```

```
   FOR J = I TO 20 +I
      PRESET (J,50)    'Löscht die Gerade
   NEXT
NEXT
```

PRINT-Anweisung

Funktion

Gibt Daten auf dem Bildschirm aus.

Syntax

PRINT[*Ausdrucksliste*][{, | ;}]

Anmerkungen

Wenn *Ausdrucksliste* nicht angegeben wird, wird eine Leerzeile ausgegeben. Wird *Ausdrucksliste* mit in die Befehlszeile aufgenommen, so werden die Werte der Ausdrücke auf dem Bildschirm ausgegeben. Die Ausdrücke in der Liste können numerische oder Zeichenkettenausdrücke sein. (Zeichenkettenliterale müssen in Anführungszeichen gesetzt werden.)

Eine gedruckte Zahl wird immer von einem Leerschritt gefolgt. Wenn die Zahl positiv ist, geht ihr ein Leerschritt voraus; wenn die Zahl negativ ist, so geht ihr ein Minuszeichen (-) voraus.

Es gibt zwei Formate, in denen **PRINT** Zahlen mit einfacher und doppelter Genauigkeit darstellt: Festkomma- und Gleitkommazahlen. Wenn **PRINT** eine Zahl einfacher Genauigkeit in einem Festkommaformat mit 7 oder weniger Ziffern ohne Verlust der Genauigkeit darstellen kann, benutzt es das Festkommaformat; andernfalls benutzt es das Gleitkommaformat. Die Zahl 1.1E-6 beispielsweise wird als .0000011 ausgegeben, während die Zahl 1.1E-7 als 1.1E-7 ausgegeben wird.

Ebenso benutzt **PRINT** das Festkommaformat, wenn es eine Zahl doppelter Genauigkeit mit 16 oder weniger Ziffern ohne Verlust der Genauigkeit im Festkommaformat darstellen kann; anderenfalls benutzt es das Gleitkommaformat. Die Zahl 1.1D-15 wird beispielsweise in der Ausgabe als .0000000000000011 dargestellt, während die Zahl 1.1D-16 als 1.1D-16 ausgegeben wird.

Die **PRINT**-Anweisung unterstützt nur die elementaren BASIC-Datentypen (Ganzzahlen, lange Ganzzahlen, reelle Zahlen einfacher Genauigkeit, reelle Zahlen doppelter Genauigkeit und Zeichenketten). Um Informationen in einen Verbund (Record) zu schreiben, benutzen Sie die **PRINT**-Anweisung mit einzelnen Verbundelementen, wie im folgenden Ausschnitt gezeigt wird:

```
TYPE MeinTyp
   Wort AS STRING*20
   Betrag AS LONG
END TYPE
DIM MeinVerb AS MeinTyp
PRINT MeinVerb.Wort
```

Ausgabepositionen

Die Position jeder ausgegebenen Größe wird durch die Zeichen bestimmt, welche die Angaben in der Liste voneinander trennen. BASIC teilt die Zeile in Ausgabezonen zu je 14 Zeichen ein. In der Liste der Ausdrücke bewirkt ein Komma, daß der nächste Wert am Anfang der nächsten Zone ausgegeben wird. Ein Semikolon bewirkt die Ausgabe des nächsten Wertes unmittelbar nach dem letzten Wert. Die Eingabe von einem oder mehreren Leerzeichen oder Tabs zwischen den Ausdrücken hat dieselbe Wirkung wie die Eingabe eines Semikolons.

Wenn ein Komma oder ein Semikolon die Liste der Ausdrücke beendet, schreibt die nächste **PRINT**-Anweisung nach Ausgabe entsprechender Leerzeichen auf derselben Zeile. Endet die Liste ohne ein Komma oder ein Semikolon, so wird am Ende der Zeile eine Wagenrücklauf-Zeilenvorschub-Folge ausgegeben. Überschreitet die Länge der ausgegebenen Zeile die Bildschirmbreite, so geht BASIC zur nächsten physikalischen Zeile über und setzt die Ausgabe dort fort.

Beispiele

In diesem Beispiel bewirken die Kommata in der `PRINT`-Anweisung, daß jeder Wert am Anfang der nächsten Ausgabezone ausgegeben wird.

```
X=5
PRINT X+5, X-5, X*(-5), X^5
END
```

Ausgabe

```
10          0          -25          3125
```

Im folgenden Beispiel bewirkt das Semikolon am Ende der ersten `PRINT`-Anweisung, daß die ersten beiden `PRINT`-Anweisungen in derselben Zeile ausgegeben werden. Die letzte `PRINT`-Anweisung bewirkt die Ausgabe einer Leerzeile vor der nächsten Anfrage.

```
DO
   INPUT "Geben Sie X ein (0 zum Beenden): ",X
   IF X = 0 THEN
      EXIT DO
   ELSE
      PRINT X "hoch 2 ist" X^2 "und";
      PRINT X "hoch drei ist" X^3
      PRINT
   END IF
LOOP
```

Ausgabe

```
Geben Sie X ein (0 zum Beenden): 9
   9 hoch 2 ist 81 und 9 hoch drei ist 729

Geben Sie X ein (0 zum Beenden): 21
  21 hoch 2 ist 441 und 21 hoch drei ist 9261

Geben Sie X ein (0 zum Beenden): 0
```

Im folgenden Beispiel bewirken die Semikolons in der `PRINT`-Anweisung, daß jeder Wert unmittelbar nach dem vorhergehenden Wert ausgegeben wird. (Denken Sie daran, daß hinter einer Zahl immer ein Leerzeichen und vor positiven Zahlen ebenfalls ein Leerzeichen steht.)

```
FOR X=1 TO 5
   J=J+5
   K=K+10
   PRINT J;K;
NEXT X
```

Ausgabe

```
 5  10  10  20  15  30  20  40  25  50
```

PRINT #-, PRINT # USING-Anweisungen

Funktion

Schreibt Daten in eine sequentielle Datei.

Syntax

PRINT # *Dateinummer*, [**USING** *Zeichenkettenausdruck*;] *Ausdrucksliste*[{, | ;}]

Anmerkungen

Dateinummer ist die Nummer, unter der die Datei zur Ausgabe geöffnet wurde. *Zeichenkettenausdruck* besteht aus den Formatierungszeichen, die unter der Anweisung **PRINT USING** beschrieben werden. Die Ausdrücke in *Ausdrucksliste* sind die numerischen und/oder Zeichenkettenausdrücke, die in die Datei geschrieben werden sollen. Leerschritte, Kommata und Semikolons in der *Ausdrucksliste* haben die gleiche Bedeutung wie in der **PRINT**-Anweisung.

Wenn Sie *Ausdrucksliste* weglassen, schreibt die **PRINT** #-Anweisung eine leere Zeile in die Datei.

PRINT # funktioniert wie **PRINT** und schreibt ein Abbild der Daten in die Datei, wie es bei einer **PRINT**-Anweisung auf dem Bildschirm angezeigt wird. Aus diesem Grund sollte auf die Begrenzung der Daten geachtet werden, damit sie richtig ausgegeben werden können.

Hinweis Wenn Kommata als Begrenzer benutzt werden, werden die Leerzeichen zwischen den Eingabefeldern ebenfalls in die Datei geschrieben.

Vergleichen Sie auch

PRINT; **PRINT USING**; **WRITE #** und Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Beispiel

Das folgende Beispiel zeigt die Effekte, die durch Auslassen und Aufnahme von Begrenzern bei mit Hilfe der **PRINT #**-Anweisung geschriebenen Daten entstehen:

```
A$ = "Kamera, Autofocus"
B$= "20.September, 1985"
C$ = "42"
Q$ = CHR$(34)
OPEN "INVENT.DAT" FOR OUTPUT AS #1 'Öffnet INVENT.DAT
                                   'zum Schreiben
'Schreibt A$, B$, C$ ohne Begrenzer:
PRINT #1, A$ B$ C$
'Schreibt A$, B$, C$ mit Begrenzer:
PRINT #1, Q$ A$ Q$ Q$ B$ Q$ Q$ C$ Q$
CLOSE #1
OPEN "INVENT.DAT" FOR INPUT AS #1 'Öffnet INVENT.DAT
                                  'zum Lesen
FOR I% = 1 TO 2              'Liest die ersten beiden
                             'Sätze und gibt sie aus
   INPUT #1, Erste$, Zweite$, Dritte$
   PRINT Erste$ TAB(30) Zweite$ TAB(60) Dritte$ : PRINT
NEXT
CLOSE #1
```

Ausgabe

```
Kamera                 Autofocus20.September   198542

Kamera, Autofocus      20.September, 1985      42
```

PRINT USING-Anweisung

Funktion

Gibt Zeichenketten oder Zahlen in einem festgelegten Format aus.

Syntax

PRINT USING *Formatzeichenkette; Ausdrucksliste*[{, | ;}]

Anmerkungen

Formatzeichenkette ist ein Zeichenkettenliteral (oder eine -variable), das die zu druckenden Zeichenkettenliterale (wie z.B. Marken) und besondere Formatzeichen enthält. Diese Zeichen bestimmen das Feld und das Format der ausgegebenen Zeichenketten oder Zahlen.

Ausdrucksliste enthält die auszugebenden Zeichenkettenausdrücke oder numerischen Ausdrücke, die durch Semikolons voneinander getrennt werden.

Formatierung von Zeichenketten

Wenn **PRINT USING** zum Schreiben von Zeichenketten verwendet wird, können Sie eines von drei Formatzeichen, die in der nachfolgenden Liste beschrieben sind, zum Formatieren des Zeichenkettenfeldes benutzen.

Zeichen	*Beschreibung*
!	Nur das erste Zeichen in der angegebenen Zeichenkette ist auszugeben.
\ \	Gibt 2 + *n* Zeichen aus der Zeichenkette aus, wobei *n* die Anzahl der Leerzeichen zwischen den beiden rückwärtigen Schrägstrichen ist. Wenn die rückwärtigen Schrägstriche ohne Leerzeichen eingegeben werden, werden zwei Zeichen ausgegeben. Mit einem Leerzeichen werden drei Zeichen ausgegeben usw. Falls die Zeichenkette länger ist als das Feld, werden die zusätzlichen Zeichen ignoriert. Ist dagegen das Feld länger als die Zeichenkette, so wird die Zeichenkette im Feld linksbündig angeordnet und rechts mit Leerzeichen aufgefüllt.
&	Zeigt ein Zeichenfeld variabler Länge an. Wenn das Feld mit dem kaufmännischen Und-Zeichen (&) angegeben wird, so wird die Zeichenkette ohne Änderung ausgegeben.

Formatierung von Zahlen

Wenn **PRINT USING** zur Ausgabe von Zahlen verwendet wird, kann das numerische Feld mit folgenden Sonderzeichen formatiert werden:

Zeichen	*Beschreibung*
#	Stellt jede Ziffernposition dar. Ziffernpositionen werden immer aufgefüllt. Wenn die auszugebende Zahl weniger Ziffern als festgelegte Positionen hat, wird die Zahl im Feld rechtsbündig angeordnet (durch Leerzeichen eingeleitet).
.	Gibt einen Dezimalpunkt aus. Ein Dezimalpunkt kann an jeder Stelle des Feldes eingefügt werden. Wenn die Formatzeichenkette festlegt, daß vor dem Dezimalpunkt eine Ziffer stehen soll, wird die Ziffer immer ausgegeben (falls erforderlich als 0). Zahlen werden nach Bedarf gerundet.
+	Bewirkt, daß das Vorzeichen der Zahl (Plus oder Minus) vor der Zahl ausgegeben wird (wenn es am Anfang der Formatzeichenkette steht) oder hinter ihr (wenn es am Ende der Formatzeichenkette steht).
-	Bewirkt, daß ein negatives Vorzeichen hinter der Zahl ausgegeben wird, wenn das Minus-Zeichen am Ende der Formatzeichenkette steht.
**	Bewirkt, daß führende Leerzeichen im numerischen Feld mit Sternen aufgefüllt werden. Zwei Sterne legen außerdem die Position für zwei weitere Ziffern fest.
$$	Bewirkt, daß direkt links vor die formatierte Zahl ein $-Zeichen gesetzt wird. Durch $$ werden zwei weitere Ziffernpositionen festgelegt; eine davon ist das Dollarzeichen.
**$	Kombiniert die Effekte der Symbole ** und $$. Führende Leerzeichen werden mit Sternen aufgefüllt, und vor der Zahl wird ein Dollarzeichen ausgegeben. Die Symbole **$ legen drei weitere Ziffernpositionen fest; eine davon ist das Dollarzeichen. Wenn negative Zahlen ausgegeben werden, steht das Minuszeichen direkt links vor dem Dollarzeichen.
,	Wenn dieses Symbol in einer Formatzeichenkette links vom Dezimalpunkt steht, bewirkt es, daß vor jeder dritten Stelle, die links vom Dezimalpunkt steht, ein Komma eingefügt wird. Steht es am Ende der Formatzeichenkette, so wird es als Teil der Zeichenkette ausgegeben. Ein Komma legt eine weitere Ziffernposition fest. Es hat keine Wirkung, wenn es mit dem Exponentialformat (^^^^ oder ^^^^^) benutzt wird.

Zeichen	*Beschreibung*
^^^^	Legt das Exponentialformat fest. Sie können auch fünf Einschaltungszeichen (^^^^^) benutzen, um große Zahlen als E+xxx darzustellen. Jede Dezimalpunktposition kann festgelegt werden. Die signifikanten Ziffern werden linksbündig angeordnet, und der Exponent wird angepaßt. Wenn kein führendes +, nachstehendes + oder kein - festgelegt ist, wird eine Ziffernposition links vom Dezimalpunkt zur Ausgabe eines Leerzeichens oder eines Minuszeichens benutzt.
_	Ein Unterstreichungszeichen in der Formatzeichenkette gibt das nächste Zeichen als Literalzeichen aus. Als Ergebnis von zwei Unterstreichungszeichen (__) in der Formatzeichenkette wird ein literales Unterstreichungszeichen ausgegeben.
%	Wenn die auszugebende Zahl größer als das festgelegte numerische Feld ist, wird vor der Zahl ein Prozentzeichen ausgegeben. Wird durch Runden der Zahl das Feld zu klein, so wird vor der gerundeten Zahl ein Prozentzeichen ausgegeben. Falls mehr als 24 Ziffern festgelegt werden, erfolgt die Fehlermeldung `Unzulässiger Funktionsaufruf.`

Beispiele

Das folgende Beispiel zeigt das Ergebnis bei Anwendung der drei Zeichenketten-Formatierungszeichen:

```
'Anwendung der drei Zeichenketten-Formatierungszeichen
'zur Änderung der Bildschirmausgabe:
A$ = "BILD" : B$ = "ZEILE"
PRINT USING "!";A$;B$
PRINT USING "\  \";A$;B$  'Zwei Leerzeichen zwischen
                          'rückwärtigen Schrägstrichen
                          ' geben 4 Zeichen von A$
                          ' aus
PRINT USING "\   \";A$;B$;"!!" 'Drei Leerzeichen -
                               'geben A$ und ein
                               'Leerzeichen aus
PRINT USING "!";A$;
PRINT USING "&";B$
```

Ausgabe

```
BZ
BILDZEIL
BILD ZEILE  !!
BZEILE
```

Das folgende Beispiel zeigt die Effekte von verschiedenen Kombinationen von numerischen Formatierungszeichen:

```
'Formatiert und gibt numerische Daten aus:
PRINT USING "##.##";.78
PRINT USING "###.##";987.654
PRINT USING "##.##   ";10.2,5.3,66.789,.234
PRINT USING "+##.##    ";-68.95,2.4,55.6,-.9
PRINT USING "##.##-    ";-68.95,22.449,-7.01
PRINT USING "**#.#    ";12.39,-0.9,765.1
PRINT USING "$$###.##";456.78
PRINT USING "**$##.##";2.34
PRINT USING "####,.##";1234.5
PRINT USING "##.##^^^^";234.56
PRINT USING ".####^^^^-";-888888
PRINT USING "+.##^^^^";123
PRINT USING "+.##^^^^^";123
PRINT USING "_!##.##_!";12.34
PRINT USING "##.##";111.22
PRINT USING ".##";.999
```

Ausgabe

```
 0.78
987.65
10.20    5.30   66.79    0.23
-68.95    +2.40   +55.60    -0.90
68.95-    22.45     7.01-
*12.4   *-0.9   765.1
 $456.78
***$2.34
```

```
1,234.50
 2.35E+02
0.8889E+06-
+.12E+03
+.12E+003
!12.34!
%111.22
%1.00
```

PSET-Anweisung

Funktion

Stellt einen Punkt auf dem Bildschirm dar.

Syntax

PSET [**STEP**] (*x-Koordinate,y-Koordinate*) [, *Farbe*]

Anmerkungen

Die folgende Liste beschreibt die Teile der Anweisung **PSET**:

Teil	*Beschreibung*
STEP	Zeigt an, daß die gegebenen *x*- und *y-Koordinaten* nicht absolut sondern relativ sind. Die Koordinaten werden als Abstände von der letzten Cursorposition und nicht als Abstände von der Bildschirmkoordinate (0,0) behandelt. Wenn beispielsweise der letzte Punkt, auf den Bezug genommen wurde, (10,10) war, würde `PSET STEP (10,5)` sich auf den Punkt (20,15) beziehen.
x-Koordinate	Die *x*-Koordinate des Bildpunktes, der gesetzt werden soll.
y-Koordinate	Die *y*-Koordinate des Bildpunktes, der gesetzt werden soll.
Farbe	Das Farbattribut für den angegebenen Bildpunkt.

Liegt eine Koordinate außerhalb des aktuellen Darstellungsfeldes, so erfolgt weder eine Aktion noch wird eine Fehlermeldung ausgegeben. **PSET** erlaubt es, in der Befehlszeile das Argument *Farbe* wegzulassen. Wenn *Farbe* nicht angegeben wird, ist die Vorgabe die Textfarbe.

Vergleichen Sie auch

PRESET

Beispiel

Dieses Beispiel zeichnet eine Gerade von (0,0) bis (100,100) und löscht diese anschließend, indem es diese Gerade mit der Hintergrundfarbe überzeichnet.

```
SCREEN 2  'Zeichnet eine Gerade von (0,0)
          'bis (100,100)
FOR I=0 TO 100
   PSET (I,I)
NEXT I
FOR I=0 TO 100      'Löscht jetzt diese Gerade
   PSET STEP (-1,-1),0
NEXT I
```

PUT-Anweisung – Datei-E/A

Funktion

Schreibt aus einer Variablen oder einem Direktzugriffspuffer in eine Datei.

Syntax

PUT [#] *Dateinummer* [,[*Satznummer*][, *Variable*]]

Anmerkungen

Die folgende Liste beschreibt die Argumente der **PUT**-Anweisung:

Argument	*Beschreibung*
Dateinummer	Die in der **OPEN**-Anweisung verwendete Nummer zum Öffnen der Datei.
Satznummer	Für Direktzugriffsdateien die Nummer des zu schreibenden Satzes. Für Dateien im Binärmodus die Byte-Position in der Datei, an der das Schreiben durchgeführt wird. Der erste Satz in einer Datei ist Satz 1. Wenn Sie *Satznummer* nicht angeben, wird in den nächsten Satz oder das nächste Byte (der/das nach dem letzten **GET** oder **PUT**, oder der/das, auf den/das das letzte **SEEK** zeigt), geschrieben. Die größtmögliche Satznummer ist 2^{31}-1 oder 2.147.483.647.
Variable	Die Variable, die die Ausgabe enthält, die in die Datei zu schreiben ist. Die Anweisung **PUT** schreibt soviele Bytes in die Datei, wie sich in der Variablen befinden.

Wenn Sie eine Variable benutzen, ist es nicht notwendig, mit **MKI$**, **MKL$**, **MKS$** oder **MKD$** vor dem Schreiben numerische Felder umzuwandeln. Sie dürfen für die Datei keine **FIELD**-Anweisung verwenden, wenn Sie das Argument *Variable* benutzen.

Für Direktzugriffsdateien können Sie jede Variable verwenden, solange die Länge dieser Variablen kleiner oder gleich der Länge des Satzes ist. Normalerweise wird eine passend zu den Feldern in einem Datensatz definierte Verbundvariable verwendet.

Für Dateien im Binärmodus können Sie jede beliebige Variable verwenden.

Wenn Sie Zeichenkettenvariablen mit variabler Länge verwenden, schreibt die Anweisung soviele Bytes, wie Zeichen in der Zeichenkette vorkommen. Zum Beispiel schreiben die folgenden zwei Anweisungen 15 Bytes in Datei Nummer 1:

```
VarZeich$=STRING$ (15,"X")
PUT #1,,VarZeich$
```

Weitere Informationen über die Verwendung von Variablen, anstelle von **FIELD**-Anweisungen für Direktzugriffsdateien, finden Sie in den Beispielen und in Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Ein Satz kann nicht mehr als 32.767 Bytes enthalten.

Sie können die *Satznummer*, die *Variable* oder beides weglassen. Wenn Sie nur die *Satznummer* weglassen, müssen Sie die Kommata weiterhin angeben:

```
PUT #4,,DateiPuffer
```

Wenn Sie beide Argumente weglassen, müssen Sie die Kommata nicht angeben:

```
PUT #4
```

Die Anweisungen **GET** und **PUT** erlauben Eingaben und Ausgaben fester Länge für BASIC-Kommunikationsdateien. Verwenden Sie **GET** und **PUT** bei Datenübertragungen mit Vorsicht, weil **PUT** eine feste Anzahl von Zeichen schreibt und bei Übertragungsfehlern unbegrenzt wartet.

Hinweis Wenn Sie einen Dateipuffer benutzen, der durch eine **FILE**-Anweisung definiert ist, können **LSET, RSET, PRINT #, PRINT # USING** und **WRITE #** verwendet werden, um Zeichen in den Direktzugriffsdatei-Puffer zu bringen, bevor eine **PUT**-Anweisung ausgeführt wird.

Bei **WRITE #** füllt BASIC den Puffer bis zum Wagenrücklaufzeichen mit Leerzeichen auf. Jeder Versuch, über das Ende des Puffers hinaus zu schreiben oder zu lesen, führt zur Fehlermeldung `FIELD-Überlauf`.

Vergleichen Sie auch

CVI, CVS, CVL, CVD, GET (Datei-E/A), LSET, MKD$, MKI$, MKL$, MKS$

Beispiel

Das folgende Beispiel liest Namen und Testergebnisse von der Tastatur ein und speichert die Namen und die Ergebnisse in einer Direktzugriffsdatei.

```
'Lies einen Namen und ein Testergebnis von der
'Tastatur ein. Speichere jeden Namen und jedes
'Testergebnis als Satz in einer Direktzugriffsdatei.

'Definiere Satzfelder.
TYPE TestSatz
   NameFeld      AS STRING*20
   ErgebnisFeld AS SINGLE
END TYPE
```

```
'Öffne die Testdaten-Datei.
DIM DateiPuffer AS TestSatz
OPEN "TESTDAT.DAT" FOR RANDOM AS #1    LEN=LEN(DateiPuffer)
'Lies Namen und dazugehörendes Ergebnis von der
'Tastatur.
I=0
DO
   I=I+1
   INPUT "Name ? ", DateiPuffer.NameFeld
   INPUT "Ergebnis ? ", DateiPuffer.ErgebnisFeld
   INPUT "Weitere Eingaben (J/N)? ",Antw$
   PUT #1, I, DateiPuffer
LOOP UNTIL UCASE$(MID$(Antw$,1,1))="N"
PRINT I;" Sätze geschrieben."
CLOSE #1
```

PUT-Anweisung – Grafik

Funktion

Zeichnet eine Grafik, die durch eine **GET**-Anweisung gespeichert wurde, auf den Bildschirm.

Syntax

PUT [**STEP**](*x,y*) *Datenfeldname*[(*Indizes*)][, *Aktionswort*]

Anmerkungen

Die Teile der **PUT**-Anweisung sind nachfolgend beschrieben:

Teil	*Beschreibung*
STEP	Zeigt an, daß die angegebenen *x*- und *y*-Koordinaten nicht absolut sondern relativ sind. Die Koordinaten werden als Abstände von der letzten Cursorposition behandelt und nicht als Abstände von der (0,0)-Bildschirmkoordinate. Wenn beispielsweise der Punkt, auf den zuletzt Bezug genommen wurde, (10,10) war, würde `PUT STEP (10,5),Ball` das in `Ball` gespeicherte Objekt bei (20,15) darstellen.

Teil	*Beschreibung*
(*x*,*y*)	Diese Koordinaten geben die obere linke Ecke des Rechtecks an, welches das Bild umschließt, das in das aktuelle Ausgabefenster gebracht werden soll.
Datenfeldname	Der Name des Datenfeldes, das das Bild enthält. Die Formel zur Berechnung der Größe dieses Feldes finden Sie unter der Anweisung **GET(Grafik)**. Das Datenfeld kann ein mehrdimensionales Datenfeld sein.
Indizes	Legt fest, daß das Bild vom angegebenen Datenfeld-Element aus reproduziert wird und nicht vom ersten Datenfeld-Element.
Aktionswort	Das *Aktionswort* bestimmt die Wechselwirkung zwischen dem gespeicherten Bild und dem bereits auf dem Bildschirm vorhandenen Bild; die verschiedenen Werte für *Aktionswort* werden in der folgenden Liste beschrieben. **XOR** ist das Standard-Aktionswort.

Wort	*Beschreibung*
PSET	Überträgt die Daten punktweise auf den Bildschirm. Jeder Punkt hat genau das Farbattribut, mit dem er mit **GET** vom Bildschirm gespeichert wurde.
PRESET	Identisch mit **PSET**, außer daß ein negatives Bild (z. B. schwarz auf weiß) erzeugt wird.
AND	Wird benutzt, wenn ein Bild über ein bereits auf dem Bildschirm vorhandenes Bild gezeichnet werden soll. Das sich ergebende Bild ist das Ergebnis eines logischen **AND** des gespeicherten Bildes und des Bildschirms; Punkte, die sowohl im gespeicherten Bild als auch im bereits existierenden Bild dieselbe Farbe haben, behalten ihre Farbe, während Punkte, welche im vorhandenen Bild und im gespeicherten Bild verschiedene Farben haben, ihre Farbe nicht behalten.
OR	Wird benutzt, um ein Bild über ein existierendes Bild zu legen. Das gespeicherte Bild löscht den vorhergehenden Bildschirminhalt nicht. Das daraus resultierende Bild ist ein Produkt des logischen **OR** des gespeicherten Bildes und des Bildschirmbildes.
XOR	Ein Spezialmodus, der oft zur Erzeugung von Animationseffekten benutzt wird. **XOR** bewirkt, daß die Punkte auf dem Bildschirm dort invertiert werden, wo im Datenfeldbild ein Punkt vorhanden ist. Dieses Verhalten entspricht genau demjenigen des Cursors: Wenn ein Bild zweimal gegen den ganzen Hintergrund ausgetauscht wird, wird der Hintergrund wiederhergestellt. Dadurch kann ein Objekt über den Bildschirm bewegt werden, ohne daß der Hintergrund gelöscht bzw. verändert wird.

Weitere detaillierte Beschreibungen zu Animation mit der Anweisung **PUT** finden Sie in Kapitel 5, "Grafiken", in *Programmieren in BASIC: Ausgewählte Themen.*

Vergleichen Sie auch

GET (Grafik), PRESET, PRINT, PSET

Beispiel

Das nachstehende Beispiel erzeugt einen sich bewegenden weißen Ball, der von den Seiten des Bildschirms zurückspringt, bis Sie eine Taste zum Beenden des Programms drücken:

```
DEFINT A-Z
DIM Ball(84) 'Dimensioniert ganzzahliges Datenfeld
             'groß genug, um den Ball aufzunehmen.
SCREEN 2     'Bildschirmauflösung 640 x 200
             'Bildpunkte.
INPUT "Beliebige Taste zum Beenden; _
       <STRG> zum Starten", Test$
CLS
CIRCLE (16,16),14 'Zeichnet den Ball und füllt
                  'ihn aus.
PAINT (16,16),1
GET (0,0)-(32,32),Ball
X = 0 : Y = 0
Xdelta = 2 : Ydelta = 1
DO
   'Weiter in derselben Richtung, solange sich der
   'Ball innerhalb der Bildschirmgrenzen (0,0) und
   '(640,200) bewegt:
   X = X + Xdelta : Y = Y + Ydelta
   IF INKEY$ <> "" THEN END 'Prüfung auf
                            'Tasteneingabe.
   'Ändere die X-Richtung, wenn der Ball die linke
   'oder rechte Bildschirmkante erreicht.
   IF (X < 1 OR X > 600) THEN
      Xdelta = -Xdelta
      BEEP
   END IF
   'Ändere die Y-Richtung, wenn der Ball die obere
   'oder untere Bildschirmkante erreicht.
   IF (Y < 1 OR Y > 160) THEN
      Ydelta = -Ydelta
      BEEP
   END IF
```

```
   'Stelle neues Bild auf dem Bildschirm dar,
   'lösche gleichzeitig das alte Bild.
   PUT (X,Y),Ball,PSET
LOOP
END
```

RANDOMIZE-Anweisung

Funktion

Initialisiert den Zufallszahlengenerator.

Syntax

RANDOMIZE [*Ausdruck*]

Anmerkungen

Wenn Sie *Ausdruck* nicht angeben, unterbricht BASIC und fordert vor der Ausführung der Anweisung **RANDOMIZE** durch folgende Anzeige einen Wert:

```
Zufallszahl setzen (-32768 bis 32767)?
```

Wenn Sie das Argument *Ausdruck* gebrauchen, benutzt QuickBASIC den Wert, um den Zufallszahlengenerator zu initialisieren.

Wenn der Zufallszahlengenerator nicht neu gesetzt wird, gibt die Funktion **RND** bei jeder Programmausführung dieselbe Folge von Zufallszahlen an. Um die Folge der Zufallszahlen bei jeder Programmausführung zu ändern, setzen Sie eine **RANDOMIZE**-Anweisung an den Anfang des Programms und ändern das Argument bei jeder Ausführung.

Eine komfortable Möglichkeit, den Zufallszahlengenerator zu initialisieren, besteht in der Benutzung der **TIMER**-Funktion. Das Benutzen von **TIMER** gewährleistet, daß bei jedem erneuten Programmlauf eine neue Serie von Zufallszahlen erzeugt wird. Vergleichen Sie hierzu das Beispiel unten.

Vergleichen Sie auch

RND, TIMER

Beispiel

Das folgende Programm simuliert das Rollen von zwei Würfeln. Die **RANDOMIZE**-Anweisung des BASIC-Programms initialisiert den Zufallszahlengenerator, so daß das Programm stets verschiedene Zahlen erzeugt.

```
'Benutze den TIMER als Grundlage für den
'Zahlengenerator.
RANDOMIZE TIMER
DO
   'Simuliere das Rollen von zwei Würfeln unter
   'Benutzung von RND
   W1=INT(RND*6)+1
   W2=INT(RND*6)+1
   'Gib das Ergebnis aus.
   PRINT "Sie haben eine";W1;"und eine";W2;
   PRINT "gewürfelt mit dem Gesamtergebnis von";W1+W2
   INPUT "Noch einmal würfeln (J/N) ?";Antw$
   PRINT
LOOP UNTIL UCASE$(MID$(Antw$,1,1))="N"
END
```

Ausgabe

```
Sie haben eine 3 und eine 5 gewürfelt mit dem Gesamtergebnis
von 8
Noch einmal würfeln (J/N) ? J
Sie haben eine 4 und eine 1 gewürfelt mit dem Gesamtergebnis
von 5
Noch einmal würfeln (J/N) ? J
Sie haben eine 5 und eine 6 gewürfelt mit dem Gesamtergebnis
von 11
Noch einmal würfeln (J/N) ? N
```

READ-Anweisung

Funktion

Liest Werte aus der **DATA**-Anweisung und weist diese den Variablen zu.

Syntax

READ *Variablenliste*

Anmerkungen

Eine *Variablenliste* ist eine Serie von gültigen BASIC-Variablen, die durch Kommata getrennt sind.

READ-Anweisungen werden immer in Verbindung mit **DATA**-Anweisungen benutzt und weisen Variablen Werte aus **DATA**-Anweisungen auf der Basis von 1:1 zu. Diese Variablen können numerische oder Zeichenkettenvariablen sein. Der Versuch, einen Zeichenkettenwert in eine numerische Variable einzulesen, produziert einen Laufzeit-Syntaxfehler. Das Lesen eines numerischen Wertes in eine Zeichenkettenvariable führt nicht zu einem Fehler, sondern speichert den Wert als Zeichenkette von Ziffern.

Werte, die in ganzzahlige Variablen eingelesen werden, werden zunächst gerundet und erst dann der Variablen zugewiesen. Das Einlesen eines zu großen numerischen Wertes in eine Variable führt zu einem Laufzeitfehler.

Zeichenkettenwerte, die in Zeichenkettenvariablen mit fester Länge eingelesen werden, werden abgeschnitten, wenn sie zu lang sind. Zeichenkettenwerte, die kürzer als die Zeichenkettenvariablen sind, werden linksbündig ausgerichtet und mit Leerzeichen aufgefüllt.

Nur einzelne Elemente einer Verbundvariablen dürfen in einer **READ**-Anweisung stehen. Siehe Beispiel.

Eine einzige **READ**-Anweisung kann eine oder mehrere **DATA**-Anweisungen (sie werden nacheinander verwendet) benutzen, oder mehrere **READ**-Anweisungen können dieselbe **DATA**-Anweisung benutzen. Wenn mehr Variablen in *Variablenliste* als Werte in der **DATA**-Anweisung oder den -Anweisungen vorhanden sind, wird die Fehlermeldung `Keine weiteren Daten vorhanden` ausgegeben. Wenn die Anzahl der angegebenen Variablen kleiner ist als die Anzahl der Elemente in der **DATA**-Anweisung oder den -Anweisungen, beginnen nachfolgende **READ**-Anweisungen mit dem Lesen von Daten beim ersten ungelesenen Element. Wenn es keine nachfolgenden **READ**-Anweisungen gibt, werden die zusätzlichen Elemente ignoriert.

Benutzen Sie die **RESTORE**-Anweisung, um **DATA**-Anweisungen erneut zu lesen.

Vergleichen Sie auch

DATA, RESTORE

Beispiel

Der folgende Ausschnitt zeigt, wie eine **READ**-Anweisung benutzt werden kann, um Informationen in den benutzerdefinierten Typ `Angestellter` einzulesen:

```
TYPE Angestellter
   Name AS STRING*35
   VersNr AS STRING*9
   TaetigNr AS INTEGER
END TYPE
DIM DieserAngst AS Angestellter
DATA "Gerd Brosky","102-89-3411",1
DATA "Peter Walter","613-06-1959",6
.
.
.
READ DieserAngst.Name, DieserAngst.VersNr,
DieserAngst.TaetigNr
.
.
.
```

Siehe auch die Beispiele zu **DATA** und **RESTORE**.

REDIM-Anweisung

Funktion

Ändert die Größe, die einem mit **$DYNAMIC** deklarierten Datenfeld zugeordnet wurde.

Syntax

REDIM [**SHARED**]*Variable*(*Indizes*)[**AS** *Typ*][,*Variable*(*Indizes*)[**AS** *Typ*]]...

Anmerkungen

Die Anweisung **REDIM** hat folgende Argumente:

Argumente	*Beschreibung*
SHARED	Das wahlfreie **SHARED**-Attribut gestattet einem Modul, Variablen mit allen Prozeduren eines Moduls zu teilen; dies unterscheidet sich von der **SHARED**-Anweisung, die nur Variablen in einem einzigen Modul betrifft. **SHARED** kann nur in einer **REDIM**-Anweisung im Modul-Ebenen-Code benutzt werden.
Variable	Ein BASIC-Variablenname.
Indizes	Die Dimensionen des Datenfeldes. Es können mehrere Dimensionen deklariert werden. Die Index-Syntax wird unten beschrieben.
AS *Typ*	Deklariert *Variable* als elementaren oder benutzerdefinierten Typ. Die elementaren Typen sind **INTEGER, LONG, SINGLE, DOUBLE** und **STRING**.

Indizes in **REDIM**-Anweisungen haben folgende Form:

[*Untergrenze* **TO**] *Obergrenze* [,[*Untergrenze* **TO**] *Obergrenze*]...

Das Schlüsselwort **TO** bietet die Möglichkeit, sowohl die Untergrenze als auch die Obergrenze eines Datenfeldindexes anzugeben. Die Argumente *Untergrenze* und *Obergrenze* sind numerische Ausdrücke, die den untersten und obersten Wert des Indexes festlegen. Unter der Beschreibung zu der **DIM**-Anweisung finden Sie weitere Informationen zum Benutzen des **TO**-Schlüsselwortes.

Die **REDIM**-Anweisung ändert die Größe, die einem mit **$DYNAMIC** deklarierten Datenfeld zugeordnet wurde. In Kapitel 2, "Datentypen", finden Sie weitere Informationen zu **$STATIC**- und **$DYNAMIC**-Datenfeldern.

Wenn eine **REDIM**-Anweisung kompiliert wird, werden alle in der Anweisung deklarierten Datenfelder als dynamische Datenfelder behandelt. Während der Programmausführung wird bei der Ausführung einer **REDIM**-Anweisung die Zuordnung für ein bereits dimensioniertes Datenfeld aufgehoben und das Datenfeld anschließend mit den neuen Dimensionen neu zugeordnet. Alte Werte von Datenfeldelementen gehen verloren, weil alle numerischen Elemente auf Null und alle Zeichenkettenelemente auf Null-Zeichenketten zurückgesetzt werden.

Hinweis Sie können die *Größe* der Dimensionen eines Datenfeldes mit **REDIM** ändern, jedoch nicht die Anzahl der Dimensionen. Zum Beispiel sind folgende Anweisungen zulässig:

```
'$DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(20,15)       'A ist immer noch
                     'zweidimensional
```

Folgende Anweisungen sind jedoch *nicht* zulässig und führen zur Kompilierzeit-Fehlermeldung `Datenfeld bereits dimensioniert`:

```
'$DYNAMIC
DIM A(50,50)
ERASE A
REDIM A(5,5,5)      'Ändert die Anzahl der
                    'Dimensionen von 2 auf 3
```

Vergleichen Sie auch

DIM, ERASE, LBOUND, OPTION BASE, SHARED, UBOUND

Beispiel

Der folgende Programmausschnitt zeigt eine Möglichkeit, **REDIM** zu verwenden, ein Verbund-Datenfeld einzurichten, wenn es benötigt wird, und später den von ihm belegten Speicherplatz wieder freizugeben.

```
TYPE SchluesselElement
   Wort AS STRING*20
   Zaehl AS INTEGER
END TYPE
'Erstelle dynamische Datenfelder.
'$DYNAMIC
.
.
.
'Richte ein Verbund-Datenfeld ein, wenn es benötigt
'wird.
REDIM Schluesselwoerter(5000) AS SchluesselElement
```

```
Schluesselwoerter(99).Wort="LÖSCHEN"
Schluesselwoerter(99).Zaehl=2
PRINT Schluesselwoerter(99).Wort,Schluesselwoerter(99).Zaehl
.
.
'Gibt den Platz, der von Schluesselwoerter eingenommen
'wird, am Ende frei.
ERASE Schluesselwoerter
.
.
.
END
```

REM-Anweisung

Funktion

Ermöglicht das Einfügen erläuternder Anmerkungen in einem Programm.

Syntax 1

REM *Anmerkung*

Syntax 2

' Anmerkung

Anmerkungen

REM-Anweisungen werden nicht kompiliert, erscheinen aber bei der Auflistung des Programms genauso, wie sie eingegeben wurden.

Sie können mit einer **GOTO**- oder **GOSUB**-Anweisung zu einer **REM**-Anweisung verzweigen. Die Ausführung wird bei der ersten ausführbaren Anweisung nach der Anweisung **REM** fortgesetzt.

Ein einzelnes Anführungszeichen kann anstelle des **REM**-Schlüsselwortes verwendet werden.

Wenn das **REM**-Schlüsselwort anderen Anweisungen in einer Zeile folgt, muß es durch Doppelpunkte von den Anweisungen getrennt werden.

Mit **REM**-Anweisungen werden außerdem Metabefehle eingeleitet (weitere Informationen hierzu finden Sie im Anhang C, "Metabefehle").

Wichtig Benutzen Sie die Form des Apostrophs für die Anweisung **REM** nicht in einer **DATA**-Anweisung, weil diese als gültiges Datenelement betrachtet wird.

Beispiele

Die beiden folgenden Zeilen sind gleichwertig (beachten Sie, daß die Doppelpunkte mit dem Anführungszeichen nicht erforderlich sind):

```
FOR I=1 TO 23 : ARRAY(I)=1 : NEXT I : REM Initialisiert _
                                          das Datenfeld.
FOR I=1 TO 23 : ARRAY(I)=1 : NEXT I  'Initialisiert
                                     'das Datenfeld.
```

RESET-Anweisung

Funktion

Schließt alle Disketten-/Plattendateien.

Syntax

RESET

Anmerkungen

Die **RESET**-Anweisung schließt alle offenen Plattendateien und schreibt alle noch im Dateipuffer befindlichen Daten auf Diskette/Platte.

Bevor eine Diskette aus dem Laufwerk entfernt wird, müssen alle Dateien geschlossen werden.

Vergleichen Sie auch

CLOSE, END, SYSTEM

RESTORE-Anweisung

Funktion

Ermöglicht es, **DATA**-Anweisungen ab einer angegebenen Zeile erneut einzulesen.

Syntax

RESTORE [{*Zeilennummer* | *Zeilenmarke*}]

Anmerkungen

Nach Ausführung einer **RESTORE**-Anweisung ohne eine angegebene *Zeilennummer* oder *Zeilenmarke* liest die nächste **READ**-Anweisung den ersten Wert in der ersten **DATA**-Anweisung des Programms ein.

Wenn *Zeilennummer* oder *Zeilenmarke* angegeben ist, liest die nächste **READ**-Anweisung den ersten Wert in der angegebenen **DATA**-Anweisung. Wenn eine Zeile angegeben ist, muß die Zeilennummer oder die Zeilenmarke im Modul-Ebenen-Code sein. (In der QuickBASIC-Umgebung werden **DATA**-Anweisungen automatisch in den Modul-Ebenen-Code bewegt.)

Vergleichen Sie auch

DATA, READ

Beispiel

Die Anweisung **RESTORE** im folgenden Fragment (aus einem Programm, das zufällige Kartensequenzen erzeugt) ermöglicht es dem Programm, die Anfrage in der Anweisung **DATA** wieder so einzulesen, daß der Benutzer neue Grenzbedingungen für die verschiedenen Farben eingeben kann:

```
DEFINT A-Z
DIM X(13), Param(5,2)
DATA Pik, Herz, Karo, Kreuz, Punkte
CALL GetParameter(Param())
DO
.
.
.
```

```
   INPUT "Mit denselben Parametern wiederholen";Ch$
   IF UCASE$(Ch$) <> "J" THEN
      INPUT "Mit neuen Parametern wiederholen";Ch$
      IF UCASE$(Ch$) <> "J" THEN
         EXIT DO
      ELSE
         RESTORE
         CALL GetParameter(Param())
      END IF
   END IF
LOOP

END

SUB GetParameter(Param(2)) STATIC
   CLS
   FOR I = 0 TO 4
      READ FARBE$
      PRINT FARBE$ " (niedrig, hoch) ";
      INPUT Param(I,0), Param(I,1)
   NEXT
END SUB
```

Ausgabe

```
Pik (niedrig, hoch)? 5,10
Herz (niedrig, hoch)? 5,10
Karo (niedrig, hoch)? 6,8
Kreuz (niedrig, hoch)? 6,8
Punkte (niedrig, hoch)? 5,15
.
.
.
Mit denselben Parametern wiederholen? n
Mit neuen Parametern wiederholen? j
Pik (niedrig, hoch)? 4,8
Herz (niedrig, hoch)? 4,8
.
.
.
```

RESUME-Anweisung

Funktion

Setzt die Programmausführung fort, nachdem eine Fehlerbehandlungsroutine ausgeführt wurde.

Syntax

RESUME [0]

RESUME NEXT

RESUME {*Zeilennummer* | *Zeilenmarke*}

Anmerkungen

Die unterschiedlichen Formen der Anweisung **RESUME** ändern den Programmfluß, wie in der folgenden Liste beschrieben:

Anweisung	*Ausführung wird fortgesetzt*
RESUME [0]	Mit der Anweisung, durch die der Fehler verursacht wurde.
RESUME NEXT	Mit der Anweisung direkt nach der Anweisung, durch die der Fehler verursacht wurde.
RESUME *Zeilennummer*	Bei *Zeilennummer*.
RESUME *Zeilenmarke*	Bei *Zeilenmarke*.

Eine **RESUME**-Anweisung, die nicht in einer Fehlerbehandlungsroutine steht, führt zur Ausgabe der Fehlermeldung `RESUME ohne Fehler`. Wenn das Ende eines Programms in einer Fehlerbehandlungsroutine erreicht wird, ohne eine **RESUME**-Prozedur gefunden zu haben, wird die Fehlermeldung `RESUME fehlt` ausgegeben.

Die Zeile, die in einer **RESUME** {*Zeilennummer* | *Zeilenmarke*}-Anweisung angegeben wird, muß auf der Modul-Ebene definiert sein. Sie sollten es sich zur Gewohnheit machen, das Benutzen von Zeilennummern oder Zeilenmarken in einer **RESUME**-Anweisung zu unterlassen. Wenn Sie keine Zeilennummer angeben, kann Ihr Programm, ohne Rücksicht darauf, wo der Fehler auftrat, fortfahren.

Hinweis Programme, die Fehlerbehandlungsroutinen enthalten, müssen entweder mit der Option **/e** (On Error) oder **/x** (Resume Next) kompiliert werden, wenn Sie von der **bc**-Befehlszeile aus kompilieren. Bei der Kompilierung in der QuickBASIC-Umgebung werden keine Optionen benötigt.

Unterschied zu BASICA

Wenn bei BASICA ein Fehler in einer **DEF FN**-Funktion auftritt, versuchen sowohl **RESUME** als auch **RESUME NEXT**, die Programmausführung in der Zeile fortzusetzen, die die Funktion enthält.

Vergleichen Sie auch

ERR, ERL, ERROR, ON ERROR

Beispiel

Dieses Beispiel enthält eine Fehlerbehandlungsroutine, die in Zeile 900 anfängt. In BASICA würde diese Routine zu einer Fehlermeldung (`Keine negativen Argumente`) führen, die für `I`=-1 und -2 ausgegeben würde. Die Routine würde dann beendet. In BASIC gibt die Fehlerbehandlungsroutine eine Meldung aus, wenn `I`=-1 ist, und setzt die Programmausführung anschließend am Anfang der **FOR...NEXT**-Schleife fort, wobei `I` auf 4 zurückgesetzt wird:

```
10 ON ERROR GOTO 900
20 DEF FNTEST(A) = 1 - SQR(A)
30 FOR I = 4 TO -2 STEP -1
40     PRINT I, FNTEST(I)
50 NEXT
60 END
900 'Fehlerbehandlungsroutine
910    PRINT "Keine negativen Argumente"
920 RESUME NEXT
```

Ausgabe

```
4       -1
3       -.7320509
2       -.4142136
1        0
0        1
```

```
-1       Keine negativen Argumente
4        -1
3       -.7320509
2       -.4142136
.
.
.
```

RETURN-Anweisung

Funktion

Gibt die Ausführungskontrolle aus einer Unterroutine zurück.

Syntax

RETURN [{*Zeilennummer* | *Zeilenmarke*}]

Anmerkungen

Ohne eine Zeilennummer oder Zeilenmarke setzt **RETURN** die Ausführung an der Stelle fort, wo ein Ereignis aufgetreten ist (für Ereignisverfolgung) oder an der **GOSUB** folgenden Anweisung (für Unterroutinenaufrufe). **GOSUB** und **RETURN** ohne Angabe einer Zeilennummer kann überall im Programm benutzt werden, wohingegen **GOSUB** und korrespondierendes **RETURN** in derselben Ebene benutzt werden müssen.

Die *Zeilennummer* oder *Zeilenmarke* in einer **RETURN**-Anweisung bewirkt einen unbedingten Rücksprung von einer **GOSUB**-Unterroutine zu der spezifizierten Zeile. **RETURN** in Verbindung mit einer Zeilenmarke oder einer Zeilennummer kann die Ausführungskontrolle nur an eine Anweisung im Modul-Ebenen-Code übergeben.

Eine **RETURN**-Anweisung kann nicht dazu benutzt werden, die Kontrolle von einem mit **SUB** definierten Unterprogramm an ein aufrufendes Programm zurückzugeben. Benutzen Sie **EXIT SUB**.

Hinweis Die BASIC-**SUB**-Prozeduren bieten eine besser strukturierte Alternative zu **GOSUB**-Unterroutinen.

Vergleichen Sie auch

EXIT, **GOSUB**, **ON** *Ereignis*

RIGHT$-Funktion

Funktion

Gibt die *n* am weitesten rechts stehenden Zeichen einer Zeichenkette an.

Syntax

RIGHT$ (*Zeichenkettenausdruck, n*)

Anmerkungen

Das Argument *Zeichenkettenausdruck* kann jede Zeichenkettenvariable, Zeichenkettenkonstante oder jeder Zeichenkettenausdruck sein. Wenn *n* gleich der Anzahl der Zeichen in *Zeichenkettenausdruck* ist (*n* = **LEN**(*Zeichenkettenausdruck*)), gibt die Funktion **RIGHT$** *Zeichenkettenausdruck* zurück. Ist *n* = 0, gibt **RIGHT$** eine Null-Zeichenkette (Länge Null) an.

Vergleichen Sie auch

LEFT$, MID$

Beispiel

```
'Dieses Programm wandelt Namen der Form:
'    Vorname [Zweiter Vorname] Nachname
'um in:
'    Nachname, Vorname [Zweiter Vorname].
'
LINE INPUT "Name: "; Nm$
I = 1 : Lzpos = 0
DO WHILE I > 0
    I = INSTR(Lzpos+1,Nm$," ") 'Bildet Position des
                               'nächsten Leerzeichens.
    IF I > 0 THEN Lzpos = I
LOOP
'Lzpos zeigt jetzt auf die Position des letzten
'Leerzeichens.
IF Lzpos = 0 THEN
   PRINT Nm$        'Gibt den Nachnamen aus.
```

```
ELSE
   'Alles nach dem letzten Leerzeichen:
   Nachname$ = RIGHT$(Nm$,LEN(Nm$)-Lzpos)
   'Alles vor dem letzten Leerzeichen:
   Vorname$ = LEFT$(Nm$,Lzpos-1)
   PRINT Nachname$ ", " Vorname$
END IF
END
```

Ausgabe

```
Name: Frank Utermoehlen
Utermoehlen, Frank
```

RMDIR-Anweisung

Funktion

Löscht ein vorhandenes Verzeichnis.

Syntax

RMDIR *Verzeichnisname*

Anmerkungen

Der *Verzeichnisname* ist der Name des Verzeichnisses, das gelöscht werden soll. Es muß eine Zeichenkette mit nicht mehr als 128 Zeichen sein. Das zu löschende Verzeichnis darf keine Dateien, bis auf das Arbeitsverzeichnis ('.') und das Stammverzeichnis ('..'), enthalten; anderenfalls erfolgt die Fehlermeldung `Verzeichnis nicht gefunden` oder `Verzeichnis-/Dateizugriffsfehler`.

RMDIR funktioniert wie der gleichnamige DOS-Befehl; die Syntax kann in BASIC jedoch nicht wie in DOS auf **RD** abgekürzt werden.

Vergleichen Sie auch

CHDIR, MKDIR

Beispiel

```
CHDIR "C:\VERKAUF\TEMP" 'Wechselt zum Unterverzeichnis
                         '\TEMP in \VERKAUF
KILL "*.*"          'Löscht alle Dateien in \TEMP
CHDIR ".."          'Wechselt zurück zu \VERKAUF
RMDIR "TEMP"        'Löscht das Unterverzeichnis \TEMP
```

RND-Funktion

Funktion

Gibt eine Zufallszahl einfacher Genauigkeit zwischen 0 und 1 an.

Syntax

RND [(*n*)]

Anmerkungen

Der Wert von *n* legt fest, wie **RND** die nächste Zufallszahl generiert:

Argument	*Angegebene Zahl*
$n < 0$	Startet immer die gleiche Folge für ein beliebig gegebenes *n*.
$n > 0$ oder *n* ausgelassen	Generiert die nächste Zufallszahl der Folge.
$n = 0$	Wiederholt die letzte generierte Zahl.

Sogar wenn *n*>0 ist, wird dieselbe Reihenfolge von Zufallszahlen bei jedem Programmlauf generiert, es sei denn, Sie initialisieren den Zufallszahlengenerator jedesmal, wenn Sie Ihr Programm starten. (Auf den Handbuchseiten unter der Beschreibung zu **RANDOMIZE** finden Sie weitere Informationen zum Initialisieren des Zufallszahlengenerators.)

Zur Erzeugung von ganzzahligen Zufallszahlen innerhalb eines gegebenen Bereichs dient folgende Formel:

INT((*Obergrenze* - *Untergrenze* + **1**)***RND** + *Untergrenze*)

Dabei ist *Obergrenze* die höchste Zahl im Bereich und *Untergrenze* die niedrigste Zahl im Bereich.

Beispiel

Siehe Beispiele zu den Anweisungen **RANDOMIZE** und **TYPE**.

RSET-Anweisung

Funktion

Bringt Daten aus dem Speicher in einen Direktzugriffsdatei-Puffer (als Vorbereitung auf eine **PUT**-Anweisung) oder ordnet den Wert einer Zeichenkette in einer Zeichenkettenvariablen rechtsbündig an.

Syntax

RSET *Zeichenkettenvariable=Zeichenkettenausdruck*

Anmerkungen

Die *Zeichenkettenvariable* ist normalerweise ein Direktzugriffsdatei-Feld, das in einer **FIELD**-Anweisung definiert ist, dennoch kann sie jede Zeichenkettenvariable sein. Der *Zeichenkettenausdruck* ist der Wert, der der Variablen zugewiesen wird.

Wenn *Zeichenkettenausdruck* weniger Bytes als in der **FIELD**-Anweisung für *Zeichenkettenvariable* definiert wurden benötigt, ordnet die **RSET**-Funktion die Zeichenkette rechtsbündig in dem Feld an (**LSET** richtet die Zeichenkette linksbündig aus). Die zusätzlichen Stellen werden mit Leerzeichen aufgefüllt. Falls die Zeichenkette zu lang für das Feld ist, schneiden sowohl **LSET** als auch **RSET** Zeichen von rechts ab. Numerische Werte müssen in Zeichenketten umgewandelt werden, bevor sie mit **LSET** oder **RSET** bündig angeordnet werden können.

Die **RSET**-Anweisung kann mit Zeichenkettenvariablen, die nicht mit **FIELD**-Anweisungen verbunden sind, benutzt werden. Bei Verwendung zusammen mit einer Variablen fester Länge wird der Wert rechtsbündig ausgerichtet und nach links mit Leerzeichen aufgefüllt.

Wenn **RSET** mit einer Zeichenkette variabler Länge verwendet wird, wird die Zeichenkette als festes Feld behandelt. Die Länge des Feldes ist die Länge des Wertes der Variablen, den sie vor der **RSET**-Anweisung hatte. Sehen Sie hierzu das Beispiel unten.

Vergleichen Sie auch

FIELD, LSET, MKD$, MKI$, MKL$, MKS$, PUT

Beispiel

Das folgende Beispiel zeigt die Wirkung des Gebrauchs von **RSET** bei der Zuordnung von Werten zu Zeichenketten variabler und fester Länge.

```
DIM TmpStr2 AS STRING * 12
PRINT "         1         2         3"
PRINT "123456789012345678901234567890"
'Benutze RSET für leere Zeichenketten variabler
'Länge.
'Nichts wird gedruckt, da TmpStr$ ein Feld mit der
'Länge Null ist.
TmpStr$ = ""
RSET TmpStr$ = "Andere"
PRINT TmpStr$
'Benutze RSET für Zeichenketten variabler Länge mit
'einem Wert.
TmpStr$ = SPACE$(20)
RSET TmpStr$ = "Andere"
PRINT TmpStr$
'Benutze RSET für Zeichenketten fester Länge.
RSET TmpStr2 = "Andere"
PRINT TmpStr2
```

Ausgabe

```
         1         2         3
123456789012345678901234567890

             Andere
    Andere
```

RTRIM$-Funktion

Funktion

Gibt eine Zeichenkette aus, deren nachfolgende (rechte) Leerzeichen entfernt sind.

Syntax

RTRIM$ (*Zeichenkettenausdruck*)

Anmerkungen

Zeichenkettenausdruck kann jeder Zeichenkettenausdruck sein.

Die Funktion **RTRIM$** arbeitet sowohl mit Zeichenketten fester als auch mit Zeichenketten variabler Länge.

Vergleichen Sie auch

LTRIM$

Beispiel

Das Beispiel zeigt die Wirkung der **RTRIM$**-Funktion auf Zeichenketten variabler und fester Länge. Sehen Sie hierzu auch das Beispiel zu der **LTRIM$**-Funktion.

```
DIM FixStr AS STRING * 10
PRINT "         1         2"
PRINT "12345678901234567890"
FixStr = "Schnur"
PRINT FixStr + "*"
PRINT RTRIM$(FixStr) + "*"
VarStr$ = "geflochten" + SPACE$(7)
PRINT VarStr$ + "*"
PRINT RTRIM$(VarStr$) + "*"
```

Ausgabe

```
         1         2
12345678901234567890
Schnur    *
Schnur*
geflochten       *
geflochten*
```

RUN-Anweisung

Funktion

Startet das augenblicklich im Speicher befindliche Programm erneut oder führt ein bestimmtes Programm aus.

Syntax

RUN [{*Zeilennummer* | *Dateiangabe*}]

Anmerkungen

Das Argument *Zeilennummer* legt die Zeile fest, an der die Ausführung beginnt. Wenn *Zeilennummer* nicht angegeben ist, beginnt die Ausführung mit der ersten ausführbaren Zeile. Diese Zeile darf sich nicht in einer BASIC **SUB**- oder **FUNCTION**-Definition befinden; es muß eine Zeile des Modul-Ebenen-Codes sein. Beachten Sie, daß die Zeile anhand einer Nummer identifiziert wird.

Das Argument *Dateiangabe* ist ein Zeichenkettenausdruck, der eine Datei angibt, die in den Speicher geladen und gestartet werden soll. Die Anweisung **RUN** löscht das augenblicklich im Speicher befindliche Programm, bevor das angegebene Programm ausgeführt wird.

Wenn Sie in der QuickBASIC-Umgebung arbeiten, ist die Standard-Erweiterung für *Dateiangabe* **.BAS**. Wenn Sie Programme außerhalb der QuickBASIC-Umgebung starten, ist die Standard-Erweiterung **.EXE**. Zum Beispiel, wenn `BINGO` der Name einer ausführbaren Datei ist, so führt die Anweisung

```
RUN "BINGO"
```

das Programm in der Datei `BINGO.EXE` aus, wenn Sie sich außerhalb der Umgebung befinden, und das Programm `BINGO.BAS`, wenn Sie sich innerhalb der Umgebung befinden.

Hinweis In Programmen, die mit **bc** kompiliert wurden, kann ein *Dateiangabe*-Argument mit **RUN** in anderen Sprachen erstellte **.EXE**-Dateien aufrufen. Kompilierte Programme (**.EXE**-Dateien) können **.BAS**-Quelldateien nicht direkt ausführen.

Da eine Zeilennummer einer **RUN**-Anweisung sich auf eine Zeile im Modul-Ebenen-Code beziehen muß, kann nur das **RUN** *Dateiangabe*-Format dieser Anweisung in einer **SUB** oder **FUNCTION** benutzt werden.

Die Anweisung **RUN** schließt alle geöffneten Dateien und löscht den aktuellen Inhalt des Speichers, bevor das gewünschte Programm geladen wird. Der Compiler unterstützt nicht die BASICA **R**-Option, die alle geöffneten Datendateien geöffnet hält. Wenn Sie eine neue Datei starten, aber alle bereits geöffneten Datendateien geöffnet lassen wollen, verwenden Sie **CHAIN**.

Vergleichen Sie auch

CHAIN

Beispiel

Das nachstehende Beispiel zeigt, wie **RUN** *Zeilennummer* alle numerischen Variablen auf Null zurücksetzt: Während die Zeilennummer nach RUN in den Zeilen 60, 70, 80 und 90 erhöht wird, verlieren die Variablen in den vorhergehenden Anweisungen ihre zugewiesenen Werte.

```
10 A = 9
20 B = 7
30 C = 5
40 D = 4
50 PRINT A,B,C,D
60 IF A = 0 THEN 70 ELSE RUN 20
70 IF B = 0 THEN 80 ELSE RUN 30
80 IF C = 0 THEN 90 ELSE RUN 40
90 IF D = 0 THEN END ELSE RUN 50
```

Ausgabe

```
9       7       5       4
0       7       5       4
0       0       5       4
0       0       0       4
0       0       0       0
```

SADD-Funktion

Funktion

Gibt die Adresse des spezifierten Zeichenkettenausdrucks an.

Syntax

SADD (*Zeichenkettenvariable*)

Anmerkungen

Die **SADD**-Funktion gibt die Adresse eines Zeichenkettenausdrucks als Offset (Near Pointer) aus dem aktuellen Datensegment an. Der Offset ist eine 2-Byte-Ganzzahl. **SADD** wird am häufigsten in Programmen verschiedener Programmiersprachen benutzt.

Das Argument kann eine einfache Zeichenkettenvariable oder ein einzelnes Element eines Zeichenketten-Datenfeldes sein. Sie dürfen keine Zeichenketten fester Länge als Argument benutzen.

Sie sollten bei der Verwendung dieser Funktion vorsichtig sein, da sich Zeichenketten im Zeichenkettenraum jederzeit verschieben können. Ursachen für Speicherverschiebungen finden Sie im Abschnitt 2.3.3, "Speicherzuweisung für Variablen".

SADD arbeitet nur mit in **DGROUP** gespeicherten Zeichenkettenvariablen. Weitere Informationen über die Abspeicherung von Variablen in BASIC finden Sie im Abschnitt 2.3.3, "Speicherzuweisung für Variablen".

Wichtig Fügen Sie keine Zeichen an das Ende oder den Beginn einer Zeichenkette, die unter Verwendung von **SADD** und **LEN** übergeben wird. Das Hinzufügen von Zeichen könnte zu Laufzeitfehlern führen.

Vergleichen Sie auch

CALL, CALLS (Nicht-BASIC); DECLARE (Nicht-BASIC); FRE; PEEK; POKE; VARPTR; VARPTR$; VARSEG

Beispiel

Das folgende Beispiel benutzt **SADD** und **LEN**, um Zeichenketten einer in C geschriebenen Funktion zu übergeben. Die C-Funktion gibt den ASCII-Wert des Zeichens an einer gegebenen Stelle der Zeichenkette an. Das C-Programm würde separat kompiliert und in einer Quick-Bibliothek gespeichert oder explizit gebunden, um eine **.EXE**-Datei zu bilden. Beachten Sie, daß **BYVAL** die Voreinstellung für C ist. Im Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC* finden Sie weitere Informationen zum Programmieren in verschiedenen Sprachen.

```
'Übergib eine Zeichenkette an eine C-Funktion unter
'Verwendung von SADD und LEN.
DEFINT A-Z
'Deklariere die Funktion.
DECLARE FUNCTION MeinAsc _
    CDECL (BYVAL A AS INTEGER, BYVAL B AS INTEGER, _
    BYVAL C AS INTEGER )
A$="abcdefghijklmnopqrstuvwxyz"
PRINT "Geben Sie eine Zeichenposition (1-26) ein"
PRINT " Geben Sie 0 zum Verlassen ein."
DO
'Lies eine Zeichenposition.
   INPUT N
'Beende, wenn Position kleiner als Null.
   IF N<=0 THEN EXIT DO
'Rufe die C-Funktion auf. Die Funktion gibt den ASCII-
'Code des Zeichens an der Position N in A$ an.
   AscWert=MeinAsc(SADD(A$),LEN(A$),N)
   PRINT "ASCII-Wert: ";AscWert;"Zeichen:
   ";CHR$(AscWert)
LOOP
END
/* C-Funktion, um den ASCII-Wert des Zeichens an der
Position pos in der Zeichenkette c der Länge len,
anzugeben.      */
int far meinasc(c,len,pos)
char near *c;
int len, pos;
{
   if(pos>len) return(c[--len]);/*Vermeide Überschreiten des
höchsten Indexes */
   else if (pos<1) return (c[0]);/*Vermeide
```

```
Unterschreiten des niedrigsten Indexes */
   else
      return(c[--pos]);  /*Position ist gültig. Gib das
Zeichen an pos-1 aus, da C-Datenfelder (Zeichenketten)
mit 0 indexiert sind.  */
}
```

Ausgabe

```
Geben Sie eine Zeichenposition (1-26) ein.
Geben Sie 0 zum Verlassen ein.
? 24
ASCII-Wert:  120 Zeichen: x
? 0
```

SCREEN-Anweisung

Funktion

Setzt die Bildschirmattribute.

Syntax

SCREEN [*Modus*][,[*Farbschalter*]][,[*A-Seite*]][,[*V-Seite*]]

Anmerkungen

Die **SCREEN**-Anweisung wird benutzt, um für eine spezielle Kombination von Bildschirm und Adapter einen passenden Bildschirmmodus zu wählen. Spätere Abschnitte erläutern die für bestimmte Adapter verfügbaren Modi. Die folgende Liste beschreibt die Argumente der **SCREEN**-Anweisung.

Argument	***Beschreibung***
Modus	Eine ganzzahlige Konstante oder ein ganzzahliger Ausdruck, der den Bildschirmmodus angibt. Die gültigen Modi werden weiter unten für die jeweiligen Adapter beschrieben.
Farbschalter	Legt fest, ob Farbe auf dem Farbmonitor bzw. Farb-TV angezeigt wird oder nicht. Der *Farbschalter* ist ein numerischer Ausdruck im Bereich von 0 bis 255. Wenn der Ausdruck wahr (Nicht-Null) ist, ist die Farbe ausgeschaltet, und es wird nur schwarz-weiß dargestellt. Wenn *Farbschalter* falsch (Null) ist, wird in Farbe dargestellt. Die Bedeutung des *Farbschalter*-Arguments kehrt sich im Bildschirmmodus 0 um. Im Bildschirmmodus 2 und höher wird der *Farbschalter* ignoriert.
A-Seite	Ein numerischer Ausdruck, der die Nummer der Bildschirmseite angibt, in die die jeweiligen Grafikanweisungen schreiben. Die nachfolgende Auflistung gibt die gültigen Werte für *A-Seite* mit verschiedenen Adaptern an.
V-Seite	Ein numerischer Ausdruck, der die Nummer der Bildschirmseite angibt, die angezeigt wird. Die nachfolgende Auflistung gibt die gültigen Werte für *V-Seite* mit verschiedenen Adaptern an.

Die nächsten zwei Abschnitte beinhalten eine Zusammenfassung der jeweiligen Bildschirmmodi und erläutern die Modi, die mit speziellen Kombinationen von Adaptern und Bildschirmen verfügbar sind. Der letzte Abschnitt faßt die Attribute und Farben zusammen.

Zusammenfassung der Bildschirmmodi

Die folgenden Absätze fassen kurz jeden der Bildschirmmodi zusammen. Die Grafikadapter, auf die hier Bezug genommen werden, sind die IBM Farb-Grafikkarte (Color Graphics Adapter, CGA), der IBM Erweiterte Grafikadapter (Enhanced Graphics Adapter, EGA), die IBM Video-Grafikkarte (Video Graphics Array, VGA) und die IBM Mehrfarben-Grafikkarte (Multicolor Graphics Array, MCGA). Detaillierte Informationen über die Wirkung eines Bildschirmmodus auf eine spezielle Kombination von Bildschirm und Adapter finden Sie im nächsten Abschnitt.

Hinweis Viele Bildschirmmodi unterstützen mehr als eine Kombination von Zeilen und Spalten auf dem Bildschirm. Unter der Beschreibung zu der **WIDTH**-Anweisung finden Sie weitere Informationen über das Verändern der Zeilen- und Spaltenzahl auf dem Bildschirm.

Screen 0

- Nur Textmodus
- Textformat entweder 40 x 25, 40 x 43, 40 x 50, 80 x 25, 80 x 43 oder 80 x 50 mit 8 x 8-Matrix (8 x 14, 9 x 14 oder 9 x 16 mit EGA- oder VGA-Adapter)
- Zuweisung von 16 Farben zu jedem der 2 Attribute
- Zuweisung von 16 Farben zu jedem der 16 Attribute (mit EGA)

Screen 1

- 320 x 200 Bildpunkte, mittlere Auflösung im Grafikmodus
- 40 x 25-Textformat mit 8 x 8-Matrix
- Zuweisung von 16 Farben zu 4 Attributen mit EGA oder VGA
- Unterstützt CGA, EGA, VGA und MCGA

Screen 2

- 640 x 200 Bildpunkte, hohe Auflösung im Grafikmodus
- 80 x 25-Textformat mit 8 x 8-Matrix
- Zuweisung von 16 Farben zu 2 Attributen mit EGA oder VGA
- Unterstützt CGA, EGA, VGA und MCGA

Screen 7

- 320 x 200 Bildpunkte, mittlere Auflösung im Grafikmodus
- 40 x 25-Textformat mit 8 x 8-Matrix
- Mehrere Bildschirmseiten
- Zuweisung von 16 Farben zu jedem der 16 Attribute
- EGA oder VGA erforderlich

Screen 8

- 640 x 200 Bildpunkte, hohe Auflösung im Grafikmodus
- 80 x 25-Textformat mit 8 x 8-Matrix
- Mehrere Bildschirmseiten
- Zuweisung von 16 Farben zu jedem der 16 Attribute
- EGA oder VGA erforderlich

Screen 9

- 640 x 350 Bildpunkte, erweiterte Auflösung im Grafikmodus
- 80 x 25- oder 80 x 43-Textformat mit 8 x 14- oder 8 x 8-Matrix
- Zuweisung von entweder 64 Farben zu 16 Attributen (mehr als 64K EGA- oder VGA-Speicher) oder 16 Farben zu 4 Attributen (64K EGA- oder VGA-Speicher)
- Mehrere Bildschirmseiten
- EGA oder VGA erforderlich

Screen 10

- 640 x 350 Bildpunkte, erweiterte Auflösung im Grafikmodus
- 80 x 25- oder 80 x 43-Textformat mit 8 x 14- oder 8 x 8-Matrix
- Mehrere Bildschirmseiten
- Zuweisung von bis zu 9 Pseudofarben zu 4 Attributen (siehe Tabellen im nächsten Abschnitt)
- EGA oder VGA erforderlich

Screen 11

- 640 x 480 Bildpunkte, sehr hohe Auflösung im Grafikmodus
- 80 x 30- oder 80 x 60-Textformat mit 8 x 16- oder 8 x 8-Matrix
- Zuweisung von bis zu 256K Farben zu 2 Attributen
- VGA oder MCGA erforderlich

Screen 12

- 640 x 480 Bildpunkte, sehr hohe Auflösung im Grafikmodus
- 80 x 30- oder 80 x 60-Textformat mit 8 x 16- oder 8 x 8-Matrix
- Zuweisung von bis zu 256K Farben zu 16 Attributen
- VGA erforderlich

Screen 13

- 320 x 200 Bildpunkte, mittlere Auflösung im Grafikmodus
- 40 x 25-Textformat mit 8 x 8-Matrix
- Zuweisung von bis zu 256K Farben zu 256 Attributen
- VGA oder MCGA erforderlich

Bildschirmmodi, Adapter und Bildschirme

Die folgenden Ausführungen beschreiben die Bildschirmmodi für bestimmte Kombinationen von Adaptern und Bildschirmen.

Der IBM Monochrom Bildschirm- und Drucker-Adapter (MDPA) kann nur mit einem monochromen Bildschirm benutzt werden. Es kann nur **SCREEN 0** (Textmodus) mit dem MDPA benutzt werden. Die folgende Tabelle faßt die Wirkung der Verwendung von **SCREEN 0** mit einem MDPA zusammen.

Modus	*Zeilen und Spalten*	*Attribute*	*Farben*	*Auflösung*	*Seiten*
0	80x25	16	3	720x350	1

Die IBM Farb-Grafikkarte (CGA) und Farbbildschirm sind typisch miteinander verbunden. Diese Kombination gestattet die Ausführung von Programmen im Textmodus und von Grafikprogrammen sowohl mit mittlerer als auch mit hoher Auflösung.

Die folgende Tabelle beschreibt die Bildschirmmodi, die mit der CGA zur Verfügung stehen.

Modus	*Zeilen und Spalten*	*Farben*	*Auflösung*	*Seiten*
0	40x25	16	320x200	8
	80x25	16	640x200	4
1	40x25	4	320x200	1
2	80x25	2	640x200	1

Der IBM Erweiterter Grafikadapter (EGA) kann entweder mit dem IBM-Farbbildschirm oder dem Erweiterten Farbbildschirm kombiniert werden. In den Modi 0, 1, 2, 7 und 8 erzeugen diese Kombinationen ähnliche Resultate, mit Ausnahme der folgenden möglichen Unterschiede:

1. Die Randfarbe kann nicht bei einem Erweiterten Farbbildschirm gesetzt werden, wenn er sich im Textmodus 640 x 350 befindet.
2. Die Textqualität ist bei dem Erweiterten Farbbildschirm besser (eine 8 x 14-Matrix für Erweiterten Farbbildschirm gegenüber einer 8 x 8-Matrix für Farbbildschirm).

Der Modus 9 nutzt die Fähigkeiten des Erweiterten Farbbildschirms vollständig aus. Modus 9 erlaubt die höchste Auflösung, die mit der Kombination EGA/Erweiterter Farbbildschirm möglich ist. Programme, die für diesen Modus geschrieben sind, werden auf keiner anderen Hardware-Konfiguration, außer der VGA, laufen.

Die folgende Tabelle faßt die Bildschirmmodi, die bei der Verwendung einer EGA verwendet werden können, zusammen.

Modus	*Zeilen und Spalten*	*Bildschirm*[a]	*Attribute*	*Farben*	*Auflösung*	*Seitengröße*	*Seiten*[b]
0	40x25	C	16	16	320x200	N/A	8
	40x25	E	16	64	320x350	N/A	8
	40x43	E	16	64	320x350	N/A	8
	80x25	C	16	16	640x200	N/A	8
	80x25	E	16	64	640x350	N/A	8
	80x25	C	16	16	640x200	N/A	8
	80x25	M	16	3	720x350	N/A	8[d]
	80x43	E	16	64	640x350	N/A	4
	80x43	M	16	3	720x350	N/A	4[d]
1	40x25	N/A	4	16	320x200	16K	1
2	80x25	N/A	2	16	640x200	16K	1
7	40x25	N/A	16	16	320x200	32K	[b]
8	80x25	N/A	16	16	640x200	64K	[b]
9[e]	80x25	E	4	64	640x350	64K	1
	80x43	E	4	64	640x350	64K	"
	80x25	E	16	64	640x350	128K	[b]
	80x43	E	16	64	640x350	128K	[b]
10	80x25	M	4	9	640x350	64K	[c]
	80x43	M	4	9	640x350	64K	[c]

[a] C = Farbbildschirm, E = Erweiterter Farbbildschirm, M = Monochromer Bildschirm, N/A = nicht anwendbar (entweder Farbbildschirm oder Erweiterter Farbbildschirm).
[b] Seiten = Bildschirmspeicher/Seitengröße. 8 Seiten Maximum. Eine Seite Minimum.
[c] Seiten = Bildschirmspeicher/2/Seitengröße. 8 Seiten Maximum. Eine Seite Minimum.
[d] Seitenanzahl ist halb so groß mit 64K.
[e] Die ersten zwei Einträge unter Modus 9 sind für eine EGA mit 64K Bildschirmspeicher. Die nächsten beiden Einträge gehen von mehr als 64K Bildschirmspeicher aus.

Nur die EGA und VGA können mit dem IBM Monochrom-Bildschirm kombiniert werden, um Programme im Modus 10 auszuführen. Dieser Modus kann benutzt werden, um monochrome Grafiken mit sehr hoher Auflösung darzustellen.

Die folgenden zwei Tabellen fassen die Standardattribute und Farben des Bildschirmmodus 10 bei der Verwendung eines monochromen Bildschirms zusammen:

Attributwert	*Angezeigte Pseudofarbe*
0	Aus
1	Ein, normale Helligkeit
2	Blinken
3	Ein, verstärkte Helligkeit

Farbwert	*Angezeigte Pseudofarbe*
0	Aus
1	Blinken, aus auf ein
2	Blinken, aus auf verstärkte Helligkeit
3	Blinken, ein auf aus
4	Ein
5	Blinken, ein auf verstärkte Helligkeit
6	Blinken, verstärkte Helligkeit auf aus
7	Blinken, verstärkte Helligkeit auf ein
8	Verstärkte Helligkeit

Die IBM Video-Grafikkarte (VGA) bietet entscheidend erweiterte Text- und Grafikdarstellungen in allen Modi. Die folgende Tabelle faßt die Modi zusammen, die mit der VGA zur Verfügung stehen.

Modus	*Zeilen und Spalten*	*Attribute*	*Farben*	*Auflösung*	*Seitengröße*	*Seiten*
0	40x25	16	64	360x400	N/A	8
	40x43	16	64	320x350	N/A	8
	40x50	16	64	320x400	N/A	4
	80x25	16	64	720x400	N/A	8
	80x43	16	64	640x350	N/A	4
	80x43	16	3	720x350	N/A	4
	80x50	16	64	640x400	N/A	4
	80x50	16	3	720x400	N/A	4
1	40x25	4	16	320x200	16K	1
2	80x25	2	16	640x200	16K	1

Modus	*Zeilen und Spalten*	*Attri-bute*	*Farben*	*Auf-lösung*	*Seiten-größe*	*Seiten*
7	40x25	16	16	320x200	32K	a
8	80x25	16	16	640x200	64K	a
9	80x25	16	64	640x350	128K	a
	80x43	16	64	640x350	128K	a
10	80x25	4	9	640x350	64K	u[b]
	80x43	4	9	640x350	64K	1
11	80x30	2	256K	640x480	64K	1
	80x60	2	256K	640x480	64K	1
12	80x30	16	256K	640x480	256K	1
	80x60	16	256K	640x480	256K	1
13	40x25	256	256K	320x200	64K	1

[a] Seiten = Bildschirmspeicher/Seitengröße. In beiden Fällen liegt das Maximum bei 8 Seiten.
[b] Seiten = Bildschirmspeicher/2/Seitengröße. In beiden Fällen liegt das Maximum bei 8 Seiten.

Siehe den Eintrag für die **PALETTE**-Anweisung zur Berechnung der Farbwerte mit der VGA.

Die IBM Mehrfarben-Grafikkarte (MCGA) kombiniert die Modi der CGA mit der sehr hohen Auflösung und dem 256K-Farbmodus der VGA, um erweiterte Text- und Grafikdarstellung in allen Modi zu unterstützen. Die folgende Tabelle faßt die Modi zusammen, die von der MCGA unterstützt werden:

Modus	*Zeilen und Spalten*	*Attri-bute*	*Farben*	*Auf-lösung*	*Seiten-größe*	*Seite*
0	40x25	16	N/A	320x400	N/A	8
	80x25	16	N/A	640x400	N/A	8
1	40x25	4	N/A	320x200	16K	1
2	80x25	2	N/A	640x200	16K	1
11	80x30	2	256K	640x480	64K	1
	80x60	2	256K	640x480	16K	1
13	40x25	256	256K	320x200	64K	1

Die MCGA benutzt die gleichen Farbwerte wie die VGA. Siehe den Eintrag für die **PALETTE**-Anweisung zur Berechnung der Farbwerte mit der MCGA.

Attribute und Farben

Es existieren verschiedene Attribute und Farbwahlmöglichkeiten für verschiedene Bildschirmmodi und Bildschirm-Hardware-Konfigurationen. (Vergleichen Sie hierzu die Nachschlageseiten der **PALETTE**-Anweisung zur Beschreibung von Attribut und Farbnummer.) Die Mehrzahl dieser Attribut- und Farbkonfigurationen werden in der folgenden Tabelle beschrieben.

Attribute für Modus			*Farbbildschirm*		*Monochromanzeige*	
1,9[a]	*2,11*	*0,7,8,9[b],12,13*	*Nummer[c]*	*Farbe*	*Nummer[d]*	*Farbe*
0	0	0	0	Schwarz	0	Aus
		1	1	Blau		Unter-strichen[e]
		2	2	Grün	1	Ein[e]
		3	3	Kobaltblau	1	Ein[e]
		4	4	Rot	1	Ein[e]
		5	5	Violett	1	Ein[e]
		6	6	Braun	1	Ein[e]
		7	7	Weiß	1	Ein[e]
		8	8	Grau	0	Aus
		9	9	Hellblau		Verstärkte Helligkeit unterstrichen
		10	10	Hellgrün	2	Verstärkte Helligkeit
1		11	11	Hellkobaltblau	2	Verstärkte Helligkeit
		12	12	Hellrot	2	Verstärkte Helligkeit
2		13	13	Hellviolett	2	Verstärkte Helligkeit
		14	14	Gelb	2	Verstärkte Helligkeit
3	1	15	15	Verstärkte Helligkeit Weiß	0	Aus

a Mit EGA speicher ≤ 64K.
b Mit EGA-Speicher > 64K.
c EGA-Farbnummern. VGA und MCGA benutzen Farbnummern, die optische Gleichheit erzeugen.
d Nur für Modus 0 monochrom.
e Aus, wenn für Hintergrund benutzt.

Beispiel

Beispiele zu der **SCREEN**-Anweisung finden Sie in den **LINE**, **CIRCLE** und **DRAW**-Anweisungen.

SCREEN-Funktion

Funktion

Liest den ASCII-Wert eines Zeichens oder seine Farbe von der angegebenen Bildschirmposition.

Syntax

SCREEN (*Zeile, Spalte* [, *Farbkennzeichen*])

Anmerkungen

Die folgende Liste beschreibt die Argumente der **SCREEN**-Funktion:

Argument	*Beschreibung*
Zeile	Die Zeilennummer der Bildschirmposition. Die *Zeile* ist ein numerischer Ausdruck, der sich als vorzeichenlose Ganzzahl berechnet.
Spalte	Die Spaltennummer der Bildschirmposition. Die *Spalte* ist ein numerischer Ausdruck, der sich als vorzeichenlose Ganzzahl berechnet.
Farbkennzeichen	Ein numerischer Ausdruck. Wenn *Farbkennzeichen* nicht Null ist, gibt **SCREEN** die Nummer der Farbe auf der Bildschirmposition an. Wenn *Farbkennzeichen* 0 oder nicht angegeben ist, wird der ASCII-Code des Zeichens an der Stelle als Ganzzahl angegeben.

Beispiele

Wenn das Zeichen bei (10,10) A ist, gibt die Funktion im folgenden Beispiel 65, den ASCII-Code für A, zurück:

```
X=SCREEN(10,10)
```

Das nachstehende Beispiel gibt das Farbattribut des Zeichens in der oberen linken Bildschirmecke an:

```
X=SCREEN(1,1,1)
```

SEEK-Anweisung

Funktion

Setzt die Position in einer Datei zum nächsten Lesen oder Schreiben.

Syntax

SEEK [#]*Dateinummer*, *Position*

Anmerkungen

Die *Dateinummer* ist eine Ganzzahl, die in der **OPEN**-Anweisung benutzt wird, um die Datei zu öffnen.

Die *Position* ist ein numerischer Ausdruck, der anzeigt, wo das nächste Lesen oder Schreiben durchgeführt wird. Die *Position* muß im Bereich von 1 bis 2.147.483.647 (2^{31}-1) liegen. Für Dateien, die im **RANDOM**-Modus geöffnet sind, ist *Position* die Nummer eines Satzes in der Datei.

Für Dateien, die im **BINARY-**, **INPUT-**, **OUTPUT-** oder **APPEND**-Modus geöffnet sind, ist *Position* die Nummer eines Bytes vom Beginn der Datei. Das erste Byte in einer Datei ist 1. Nach einem **SEEK** beginnt die nächste Datei-E/A-Operation an diesem Byte in der Datei.

Hinweis Satznummern in einer **GET**- oder **PUT**-Anweisung überschreiben die von **SEEK** vorgenommene Dateipositionierung.

Ein **SEEK** auf eine negative Position oder auf die Position Null führt zu der Fehlermeldung `Falsche Datensatznummer`. Das Schreiben in eine Datei nach der Ausführung eines **SEEK** hinter das Dateiende erweitert die Datei.

Bei der Benutzung mit einem Gerät, das **SEEK** nicht unterstützt, ignoriert BASIC **SEEK** und läßt die Dateiposition ungeändert. Die BASIC-Geräte (**SCRN:**, **CONS:**, **KYBD:**, **COM***n***:** und **LPT***n***:**) unterstützen **SEEK** nicht.

Vergleichen Sie auch

GET (Datei-E/A); **OPEN**; **PUT (Datei-E/A)**; **SEEK- Funktion**; und Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Beispiel

Der folgende Codeausschnitt verwendet eine Kombination der **SEEK**-Funktion und der **SEEK**-Anweisung, um innerhalb einer Datei zurückzugehen und einen Satz zu überschreiben, wenn eine Variable wahr (nicht Null) ist.

```
CONST FALSCH=0, WAHR=NOT FALSCH
.
.
.
IF ueberschreiben = WAHR THEN
     ' Gehe innerhalb der Datei um die Länge der
     ' zu schreibenden Satzvariablen zurück
     SEEK #1, SEEK(1) -LEN(SatzVar)
     PUT #1,,SatzVar
END IF
```

SEEK-Funktion

Funktion

Gibt die aktuelle Dateiposition an.

Syntax

SEEK (*Dateinummer*)

Anmerkungen

Die *Dateinummer* ist die Nummer, die in der **OPEN**-Anweisung benutzt wurde, um die Datei zu öffnen. **SEEK** gibt einen Wert im Bereich von 1 bis 2.147.483.647 (das entspricht 2^{31}-1) an.

SEEK gibt bei Verwendung für Dateien im **RANDOM**-Modus die Nummer des als nächsten zu lesenden oder zu schreibenden Satzes an. Für im **BINARY-**, **OUTPUT-**, **APPEND-** oder **INPUT**-Modus geöffnete Dateien gibt **SEEK** die Byte-Position in der Datei an, an der die nächste Operation durchgeführt wird. Das erste Byte in einer Datei ist 1.

Bei der Benutzung mit einem Gerät, das **SEEK** nicht unterstützt, gibt die Funktion 0 aus. Die BASIC-Geräte (**SCRN:**, **CONS:**, **KYBD:**, **COM***n*: und **LPT***n*:) unterstützen **SEEK** nicht.

Vergleichen Sie auch

GET (Datei-E/A); **OPEN**; **PUT (Datei-E/A)**; **SEEK-Anweisung**; Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Beispiel

Das folgende Codefragment schreibt eine Meldung, die anzeigt, ob das letzte Schreiben oder Lesen im ersten, zweiten oder letzten Drittel der Datei vorgenommen wurde.

```
SELECT CASE (SEEK(1))
    CASE IS < .333*LOF(1)
       PRINT "Im ersten Drittel der Datei."
    CASE .333*LOF(1) TO .667+LOF(1)
       PRINT "Im zweiten Drittel der Datei."
    CASE IS >= .667*LOF(1)
       PRINT "Im letzten Drittel der Datei."
    CASE ELSE
END SELECT
```

SELECT CASE-Anweisung

Funktion

In Abhängigkeit von einem Ausdruck wird einer von mehreren Anweisungsblöcken ausgeführt.

Syntax

SELECT CASE *Testausdruck*
CASE *Ausdrucksliste1*
  [*Anweisungsblock-*1]
[**CASE** *Ausdrucksliste2*
  [*Anweisungsblock-2*]]
.
.
.
[**CASE ELSE**
  [*Anweisungsblock-n*]]
END SELECT

Anmerkungen

Die folgende Liste beschreibt die Teile der Anweisung **SELECT CASE**.

Argument	***Beschreibung***
Testausdruck	Jeder numerische oder Zeichenkettenausdruck.
Anweisungsblock-1, Anweisungsblock-2, Anweisungsblock-n	Die Elemente *Anweisungsblock-1* bis *Anweisungsblock-n* bestehen aus einer beliebigen Anzahl von Anweisungen auf einer oder mehreren Zeilen.
Ausdrucksliste1, Ausdrucksliste2	Diese Elemente können jede der folgenden Formen haben: *Ausdruck*[, *Ausdruck*...] *Ausdruck* **TO** *Ausdruck* **IS** *Vergleichsoperator Ausdruck*

Die folgende Liste beschreibt die Teile einer *Ausdrucksliste*:

Argument	***Beschreibung***
Ausdruck	Jeder numerische oder Zeichenkettenausdruck. Der Typ des Ausdrucks muß mit dem Typ des zu testenden Ausdrucks übereinstimmen.
Vergleichs-operator	Jeder der folgenden Operatoren:

Symbol	***Bedeutung***
<	Kleiner als
<=	Kleiner oder gleich
>	Größer als
>=	Größer oder gleich
<>	Ungleich
=	Gleich

Wenn der *Testausdruck* mit der *Ausdrucksliste*, die mit einer **CASE**-Klausel verknüpft ist, übereinstimmt, wird der Anweisungsblock, der der **CASE**-Klausel folgt, bis zur nächsten **CASE**-Klausel oder, für die letzte, bis **END SELECT** ausgeführt. Die Kontrolle wird dann der Anweisung übergeben, die **END SELECT** folgt.

Wenn Sie das Schlüsselwort **TO** verwenden, um einen Bereich von Werten anzugeben, muß der kleinere Wert an erster Stelle stehen. Zum Beispiel werden die Anweisungen, die mit `CASE -1 TO -5` verknüpft sind, nicht ausgeführt, wenn der *Testausdruck* `-4` ist. Die Zeile sollte als `CASE -5 TO -1` geschrieben sein.

Einen Vergleichsoperator können Sie nur benutzen, wenn das Schlüsselwort **IS** erscheint.

Wenn **CASE ELSE** verwendet wird, werden die verknüpften Anweisungen nur ausgeführt, wenn der *Testausdruck* mit keinem der anderen **CASE**-Auswahlen übereinstimmt. Es ist ratsam, eine **CASE ELSE**-Anweisung in einem **SELECT CASE**-Block zu verwenden, um unvorgesehene Werte für *Testausdruck* abzufangen. Wenn keine der in der Liste enthaltenen Ausdrücke in der **CASE**-Klausel mit dem *Testausdruck* übereinstimmt, erhalten Sie einen Laufzeitfehler. Zum Beispiel erscheint die Laufzeit-Fehlermeldung `CASE ELSE erwartet`, wenn der Benutzer `SECHS` als Eingabe in den folgenden Programmausschnitt gibt:

```
INPUT x$
SELECT CASE x$
    CASE "EINS"
        PRINT "Eins"
```

```
    CASE "ZWEI"
        PRINT "Zwei"
    CASE "DREI"
        PRINT "Drei"
END SELECT
```

In jeder **CASE**-Klausel können Sie mehrere Ausdrücke oder Bereiche verwenden. Zum Beispiel ist die folgende Zeile zulässig:

```
CASE 1 TO 4, 7 TO 9, 11, 13, IS > MaxZahl%
```

Für Zeichenketten können Sie ebenfalls Bereiche und mehrere Ausdrücke angeben:

```
CASE "alles", "Birnen" TO "Suppe", TestGroesse$
```

CASE sucht nach Zeichenketten, die absolut identisch sind mit `alles`, dem aktuellen Wert von `TestGroesse$`, oder in der alphabetischen Reihenfolge zwischen `Birnen` und `Suppe` liegen.

Wenn ein Ausdruck in mehreren **CASE**-Klauseln verwendet wird, werden nur die Anweisungen ausgeführt, die mit dem ersten Vorkommen des Ausdrucks verknüpft sind.

Beispiel

Im folgenden Programm wird die **SELECT CASE**-Anweisung verwendet, um, abhängig vom Eingabewert, verschiedene Aktionen zu starten:

```
'Programm demonstriert verschiedene Formen von CASE-
'Bedingungen
INPUT "Geben Sie ein zu akzeptierendes Risiko ein _
       (1- 10) : ", Total
SELECT CASE Total
   CASE IS >= 10
      PRINT "Maximales Risiko und maximaler_
             potentieller Gewinn"
      PRINT "Kaufen Sie Aktienpakete"
   CASE 6 TO 9
      PRINT "Hohes Risiko und hoher potentieller_
             Gewinn"
      PRINT "Kaufen Sie Pfandbriefe"
   CASE 2 TO 5
      PRINT "Mittleres Risiko und mittlerer Gewinn"
```

```
      PRINT "Kaufen Sie Bundesschatzbriefe"
   CASE 1
      PRINT "Kein Risiko, kleiner Gewinn"
      PRINT "Kaufen Sie Gold"
   CASE ELSE
      PRINT "Antwort außerhalb des Bereiches"
END SELECT
```

Ausgabe

```
Geben Sie ein zu akzeptierendes Risiko ein (1-10) : 10
Maximales Risiko und maximaler potentieller Gewinn
Kaufen Sie Aktienpakete

Geben Sie ein zu akzeptierendes Risiko ein (1-10) : 0
Antwort außerhalb des Bereiches
```

SETMEM-Funktion

Funktion

Verändert die Größe des Speichers, der vom Far Heap genutzt wird - der Bereich, in dem weite Objekte (Far Objects) und interne Tabellen gespeichert werden.

Syntax

SETMEM *(Numerischer Ausdruck)*

Anmerkungen

Die Funktion **SETMEM** vergrößert oder verkleinert den Far Heap mit der Anzahl von Bytes, die durch *Numerischer Ausdruck* angegeben werden. Weitere Informationen darüber, was BASIC als weite Objekte abspeichert, finden Sie in Abschnitt 2.3.3, "Speicherzuweisung von Variablen". Wenn *Numerischer Ausdruck* negativ ist, verkleinert **SETMEM** den Far Heap um die angegebene Anzahl von Bytes. Wenn *Numerischer Ausdruck* positiv ist, versucht **SETMEM**, den Bereich des Far Heap um die Anzahl von Bytes zu vergrößern. **SETMEM** gibt die Gesamtzahl von Bytes in dem Far Heap zurück. Wenn *Numerischer Ausdruck* Null beträgt, gibt **SETMEM** die aktuelle Größe des Far Heap an.

Wenn **SETMEM** den Far Heap nicht um die erforderliche Anzahl von Bytes verändern kann, ordnet es soviele Bytes wie möglich neu zu.

SETMEM kann bei der Programmierung in verschiedenen Sprachen zur Verkleinerung des Far Heap-Bereiches verwendet werden, so daß Prozeduren in anderen Sprachen Far-Speicher dynamisch zuweisen können.

Der Versuch, mit **SETMEM** als erstem Schritt den Far Heap zu vergrößern, ist wirkungslos, da BASIC beim Programmstart dem Far Heap soviel Speicher wie möglich zuweist.

Beispiel

Das folgende Programm zeigt, wie **SETMEM** benutzt werden kann, um Speicher für eine C-Funktion freizugeben, die **malloc** verwendet, um den dynamischen Speicher zu ermitteln. Die C-Funktion ist separat kompiliert und wird dann in eine Quick-Bibliothek eingebracht oder in das BASIC-Programm eingebunden. Die C-Funktion ist unter Benutzung des großen Speichermodells kompiliert, so daß Aufrufe von **malloc** den Far-Bereich benutzen, der von dem BASIC-Programm freigesetzt wurde.

```
DEFINT A-Z
DECLARE SUB CFunk CDECL (BYVAL X AS INTEGER)
'Verkleinere die Größe des Far Heap, sodaß CFunk
'malloc verwenden kann, um den dynamischen Speicher zu
'ermitteln.
VorAufruf=SETMEM(-2048)
'Aufruf der C-Funktion.
CFunk(1024%)
'Gib den Speicher an den Far Heap zurück. Benutze
'einen größeren Wert, sodaß der gesamte Bereich zurück
'an den Heap geht.
NachAufruf=SETMEM(3500)
IF NachAufruf <= VorAufruf THEN PRINT "Speicher _
                 nicht wieder zugewiesen."
END
void far cfunk(bytes)
int bytes;
{
  char *malloc();
  char *workspace;
   /* Weise Arbeitsspeicher zu; benutze hierzu den
Betrag, den BASIC freigegeben hat */
```

```
   workspace=malloc((unsigned) bytes);
   /* Arbeitsbereich wird hier benutzt */
   /* Gib Speicher frei, bevor Rücksprung nach BASIC
erfolgt */
   free(workspace);
}
```

SGN-Funktion

Funktion

Gibt das Vorzeichen eines numerischen Ausdrucks an.

Syntax

SGN(*Numerischer Ausdruck*)

Anmerkungen

Die **SGN**-Funktion gibt einen Wert aus, der abhängig vom Vorzeichen seines Arguments ist:

Wenn *Numerischer Ausdruck* > 0, dann gibt **SGN**(*Numerischer Ausdruck*) 1 an.

Wenn *Numerischer Ausdruck* = 0, dann gibt **SGN**(*Numerischer Ausdruck*) 0 an.

Wenn *Numerischer Ausdruck* < 0, dann gibt **SGN**(*Numerischer Ausdruck*) -1 an.

Beispiel

Das nachstehende Programm berechnet und druckt die Lösung für die eingegebene quadratische Gleichung (oder Gleichung zweiten Grades). Das Programm benutzt das Vorzeichen eines Testausdrucks, um festzulegen, wie die Lösung berechnet werden soll.

```
CONST NRealLoes=-1, EineLoesng=0, ZweiLoesng=1
'Gib Koeffizienten einer quadratischen Gleichung ein:
'ax^2 + bx + c = 0.
INPUT;"a = ",A
INPUT;", b = ",B
INPUT ", c = ",C
Test = B^2 - 4*A*C
```

```
SELECT CASE SGN(Test)
   CASE NRealLoes
      PRINT "Diese Gleichung hat keine Lösung im _
             reellen Zahlenbereich."
   CASE EineLoesng
      PRINT "Diese Gleichung hat eine Lösung: ";
      PRINT -B/(2*A)
   CASE ZweiLoesng
      PRINT "Diese Gleichung hat zwei Lösungen: ";
      PRINT ((-B + SQR(Test))/2*A) " und ";
      PRINT ((-B - SQR(Test))/2*A)
END SELECT
```

Ausgabe

```
a = 12, b = -5, c = -2
Diese Gleichung hat zwei Lösungen:   96    -36
```

SHARED-Anweisung

Funktion

Ermöglicht einer **SUB**- oder **FUNCTION**-Prozedur Zugriff auf im Modul-Ebenen-Code deklarierten Variablen, ohne sie als Parameter zu übergeben.

Syntax

SHARED *Variable* [**AS** *Typ*][, *Variable*[**AS** *Typ*]]...

Anmerkungen

Das Argument *Variable* ist entweder ein Datenfeldname, gefolgt von (), oder ein Variablenname. Die **AS**-Klausel kann benutzt werden, um den Typ der Variablen anzuzeigen. Das *Typ*-Argument kann entweder **INTEGER, LONG, SINGLE, DOUBLE, STRING**, eine Zeichenkette fester Länge (**STRING***Länge*) oder ein benutzerdefinierter Typ sein.

Indem Sie entweder die Anweisung **SHARED** in einer **SUB**- oder **FUNCTION**-Prozedur oder das Attribut **SHARED** mit **COMMON** oder **DIM** im Modul-Ebenen-Code verwenden, können Sie Variablen in einer Prozedur benutzen, ohne sie als Parameter zu übergeben. Das **SHARED**-Attribut teilt Variablen zwischen allen Prozeduren eines Moduls, während bei der **SHARED**-Anweisung Variablen mit einer einzelnen Prozedur und dem Modul-Ebenen-Code geteilt werden.

Hinweis Die Anweisung **SHARED** teilt nur Variablen innerhalb eines einzelnen kompilierten Moduls. Es teilt keine Variablen mit Programmen in der Quick-Bibliothek oder mit separat kompilierten und in das Programm eingebundenen Prozeduren. Die **SHARED**-Anweisung teilt Variablen nur zwischen dem Modul-Ebenen-Code und einer **SUB** oder **FUNCTION** desselben Moduls.

Die **SHARED**-Anweisung kann nur in einer **SUB** oder **FUNCTION** erscheinen.

Vergleichen Sie auch

COMMON; DIM; SUB; Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen*; Kapitel 4, "Programme und Module", in diesem Buch

Beispiel

Das nachstehende Programm ruft ein Unterprogramm mit dem Namen UMWANDL auf, das die eingegebene Dezimalzahl in ihre Zeichenkettendarstellung auf der angegebenen neuen Basis umwandelt. Die Zeichenkette Z$ wird vom Unterprogramm und Hauptprogramm gemeinsam benutzt.

```
DEFINT A-Z
DO
   INPUT "Dezimalzahl (Nummer <= 0 zum _
         Beenden):  ",Dezimal
   IF Dezimal <= 0 THEN EXIT DO
   INPUT "Neue Basis: ",Neubasis
   Z$ = ""
   PRINT Dezimal "zur Basis 10 ist gleich ";
   DO WHILE Dezimal > 0
      CALL Umwandl (Dezimal,Neubasis)
      Dezimal = Dezimal\Neubasis
   LOOP
   PRINT Z$ " zur Basis" Neubasis
   PRINT
LOOP
```

```
SUB Umwandl (D,Nb) STATIC
SHARED Z$
   'Nimm den Rest, um den Wert der aktuellen Ziffer zu
   'finden.
   R = D MOD Nb
   'Wenn die Ziffer kleiner als 10 ist, gib eine
   'Ziffer aus (0...9)
   'Andernfalls gib einen Buchstaben aus (A...Z).
   IF R < 10 THEN ZkZahl$ = CHR$(R+48) _
      ELSE ZkZahl$ = CHR$(R+55)
   Z$ = RIGHT$(ZkZahl$,1) + Z$
END SUB
```

Ausgabe

```
Dezimalzahl (Nummer <= 0 zum Beenden): 256
Neue Basis: 2
 256 zur Basis 10 ist gleich 100000000 zur Basis 2
Dezimalzahl (Nummer <= 0 zum Beenden): 31
Neue Basis: 16
 31 zur Basis 10 ist gleich 1F zur Basis 16
Dezimalzahl (Nummer <= 0 zum Beenden): -1
```

SHELL-Anweisung

Funktion

Verläßt das BASIC-Programm, führt ein **.COM-**, **.EXE-** oder **.BAT**-Programm oder einen DOS-Befehl aus und kehrt zum Programm in die Zeile nach der Anweisung **SHELL** zurück.

Syntax

SHELL [*Befehlszeichenkette*]

Anmerkungen

Die *Befehlszeichenkette* muß ein gültiger Zeichenkettenausdruck sein, der den Namen de auszuführenden Programms und jede Programmoption enthält.

Jede **.COM**-Datei, **.EXE**-Datei oder **.BAT**-Programm oder eine DOS-Funktion, das/die unter der Anweisung **SHELL** ausgeführt wird, nennt man eine "Zwischenbearbeitung" (Child Process). Zwischenbearbeitungen werden von der **SHELL**-Anweisung ausgeführt indem eine Kopie von **COMMAND.COM** geladen und mit der Option **/c** automatisch ausgeführt wird; diese Option ermöglicht es, daß alle Parameter in *Befehlszeichenkette* ar die Zwischenbearbeitung übergeben werden können. Außerdem ermöglicht sie die Umleitung von Standardein- und -ausgabe und die Ausführung von internen Befehlen wie **DIR, PATH** oder **SORT**.

Der Programmname in *Befehlszeichenkette* kann jede beliebige Erweiterung haben. Wird keine Erweiterung angegeben, so sucht **COMMAND.COM** zunächst nach einer **.COM**-Datei, anschließend nach einer **.EXE**-Datei und zuletzt nach einer **.BAT**-Datei. Falls **COMMAND.COM** nicht gefunden wird, gibt **SHELL** die Fehlermeldung `Datei nicht gefunden` aus. Wenn **COMMAND.COM** die in *Befehlszeichenkette* angegebene Datei nicht finden kann, wird keine Fehlermeldung von BASIC erzeugt.

Jeder Text, der vom Programmnamen durch mindestens ein Leerzeichen getrennt ist, wird von **COMMAND.COM** wie ein Programmparameter behandelt.

Während der Ausführung der Zwischenbearbeitungen bleibt BASIC im Speicher. Nach Beendigung der Zwischenbearbeitungen wird BASIC fortgesetzt.

SHELL ohne *Befehlszeichenkette* ergibt eine neue **COMMAND.COM**-Oberfläche. Sie können jetzt alles ausführen, was **COMMAND.COM** zuläßt. Wenn Sie zu BASIC zurückkehren wollen, geben Sie den DOS-Befehl **EXIT** ein.

Beispiele

Das folgende Beispiel zeigt, wie eine einzelne **SHELL**-Anweisung ein neue **COMMAND.COM** startet:

```
SHELL    'Neue COMMAND.COM
```

Ausgabe

```
The IBM Personal Computer DOS
Version 3.20 (C) Copyright International Business
                 Machines Corp 1981, 1986
             (C) Copyright Microsoft Corp 1981, 1986

D:QB4>
```

Das folgende Beispiel kopiert alle Dateien, die an einem bestimmten Datum geändert wurden, von einem bestimmten Verzeichnis:

```
'Dieses Programm kopiert alle Dateien in diesem
'Verzeichnis, die an einem bestimmten Datum geändert
'wurden, auf das von Ihnen angegebene Laufwerk und in
'das angegebene Verzeichnis.
DECLARE FUNCTION HolName$ (DirZeile$)
LINE INPUT "Geben Sie Ziellaufwerk und -verzeichnis _
    ein: ", VerzName$
SHELL "DIR > DIRDATEI"     'Speichert Verzeichnisauf-
                           'listung in DIRDATEI
DO
   OPEN "DIRDATEI" FOR INPUT AS #1
   INPUT "Geben Sie das Datum ein (MM-TT-JJ): ", _
          MDatum$
   PRINT
   'Lies DIRDATEI, prüfe auf eingegebenes Datum:
   DO
      LINE INPUT #1, DirZeile$
      'Prüfe Verzeichniszeile, um festzustellen, ob
      'Datum übereinstimmt, und die Zeile nicht eines
      'der Sonderverzeichnisse ist (. oder ..).
      IF INSTR(DirZeile$,MDatum$) > 0 AND _
        LEFT$    (DirZeile$,1) <> "." THEN
         DateiAng$ = HolName$(DirZeile$)
         'Stelle sicher, daß unsere temporäre Datei
         'nicht entfernt wird.
         IF DateiAng$ <> "DIRDATEI" THEN
             'Erstelle die DOS-Befehlszeile, um die
             'Datei zu kopieren.
             DoZeile$ = "COPY " + DateiAng$ + "  " + _
                                  VerzName$
             PRINT DoZeile$
             'Kopiere die Datei
             SHELL DoZeile$
         END IF
     END IF
   LOOP UNTIL EOF(1)
   CLOSE #1
   PRINT "Neues Datum?"
   R$ = INPUT$(1)
   CLS
LOOP UNTIL UCASE$(R$) <> "J"
```

```
  KILL "DIRDATEI"
END
'Diese Funktion holt den Dateinamen und seine
'Erweiterung aus der Verzeichnisauflistung.
FUNCTION HolName$ (DirZeile$) STATIC
   StammName$ = RTRIM$(LEFT$(DirZeile$,8))

   'Prüfe auf Erweiterung.
   ExtName$ = RTRIM$(MID$(DirZeile$,10,3))

   IF ExtName$ <> "" THEN
      StammName$ = StammName$ + "." + ExtName$
   END IF

   HolName$ = StammName$
END FUNCTION
```

Ausgabe

```
Geben Sie Ziellaufwerk und -verzeichnis ein: c:\bas\temp
Geben Sie das Datum ein (MM-TT-JJ): 11-6-87
COPY CONF.DAT  c:\bas\temp
COPY TEST.BAS  c:\bas\temp
COPY TEMPFILE  c:\bas\temp
Neues Datum? n
```

SIN-Funktion

Funktion

Gibt den Sinus von dem Winkel *x* an, wobei *x* im Bogenmaß steht.

Syntax

SIN(*x*)

Anmerkungen

Die **SIN**-Funktion wird mit doppelter Genauigkeit berechnet, wenn *x* ein Wert doppelter Genauigkeit ist. Wenn *x* kein Wert doppelter Genauigkeit ist, wird **SIN** mit einfacher Genauigkeit berechnet.

Vergleichen Sie auch

ATN, COS, TAN

Beispiel

Das Beispiel stellt die Grafik der Polargleichung $r = 1 + \sin n(\theta)$ dar. Wegen ihrer Ähnlichkeit mit einem Herzen, wenn *n* gleich 1 ist, wird diese Figur manchmal als "Herzkurve" bezeichnet.

```
CONST PI = 3.141593
SCREEN 1 : COLOR 1,1      'Mittlere Auflösung, blauer
                          'Hintergrund
WINDOW (-3,-2)-(3,2)      'Wandele Bildschirm in
                          'kartesische Koordinaten um
INPUT "Anzahl der Blütenblätter = ", N
CLS
PSET (1,0)       'Setzt Startpunkt
FOR Winkel = 0 TO 2*PI STEP .02
    R = 1 + SIN(N*Winkel) 'Polargleichung für "Blüte"
    X = R * COS(Winkel)    'Wandle Polarkoordinaten in
    Y = R * SIN(Winkel)    'kartesische Koordinaten um
    LINE -(X,Y)           'Zeichne eine Gerade vom
                          'alten zum neuen Punkt
NEXT
```

SOUND-Anweisung

Funktion

Erzeugt Töne über den Lautsprecher.

Syntax

SOUND *Tonhöhe, Länge*

Anmerkungen

Tonhöhe ist die gewünschte Frequenz in Hertz (Schwingungen pro Sekunde). Dies muß ein numerischer Ausdruck sein, der eine vorzeichenlose ganze Zahl im Bereich von 37 bis 32.767 angibt.

Länge ist die Länge des Tons in Zeitschlägen. (Zeitschläge erfolgen 18.2 mal pro Sekunde.) *Länge* muß ein numerischer Ausdruck sein, der eine vorzeichenlose ganze Zahl im Bereich von 0 bis 65.535 angibt.

Wenn die Länge gleich Null ist, wird der Ton einer laufenden **SOUND**-Anweisung ausgeschaltet. Wird augenblicklich keine Anweisung **SOUND** ausgeführt, so ist eine **SOUND**-Anweisung mit einer Länge Null wirkungslos.

Vergleichen Sie auch

PLAY

Beispiel

Dieser Programmausschnitt erzeugt ein Aufwärts-/Abwärts-Glissando.

```
FOR I = 440 TO 1000 STEP 5
   SOUND I, I/1000
NEXT
FOR I = 1000 TO 440 STEP -5
   SOUND I, I/1000
NEXT
```

SPACE$-Funktion

Funktion

Gibt eine Zeichenkette von *n* Leerzeichen an.

Syntax

SPACE$(*n*)

Anmerkungen

Der Ausdruck *n* wird auf eine ganze Zahl gerundet und muß im Bereich von 0 bis 32.767 liegen.

Vergleichen Sie auch

SPC

Beispiel

```
FOR I=1 TO 5
   X$=SPACE$(I)
   PRINT X$;I
NEXT I
```

Ausgabe

```
 1
  2
   3
    4
     5
```

SPC-Funktion

Funktion

Überspringt in einer **PRINT**-Anweisung *n* Leerzeichen.

Syntax

SPC(*n*)

Anmerkungen

SPC darf nur bei **PRINT**- und **LPRINT**-Anweisungen verwendet werden. Das Argument *n* muß im Bereich von 0 bis 32.767 liegen. Hinter dem Befehl **SPC**(*n*) wird ein Semikolon (;) angenommen.

Vergleichen Sie auch

SPACE$

Beispiel

```
PRINT "VON";SPC(15);"BIS"
```

Ausgabe

```
VON               BIS
```

SQR-Funktion

Funktion

Gibt die Quadratwurzel von *n* an.

Syntax

SQR(*n*)

Anmerkungen

Das Argument *n* muß größer gleich 0 sein.

Beispiel

Das folgende Programm stellt die Kurven dar von $y = \sqrt{-x}$ für $-9 <= x <0$, und $y = \sqrt{x}$ für $0 <= x <=9$:

```
SCREEN 1 : COLOR 1     'Farbgrafikmodus, niedrige
                       'Auflösung
WINDOW (-9, -.25)-(9,3.25) 'Wandelt Bildschirm-
                           'Koordinaten in kartesische
                           'Koordinaten um.
LINE (-9,0)-(9,0)          'Zeichnet X-Achse.
LINE (0,-.25)-(0,3.25)     'Zeichnet Y-Achse.
```

```
FOR X = -9 TO 9
   LINE (X,.04)-(X,-.04)    'Zeichnet Einteilung der X-
                            'Achse.
NEXT
FOR Y = .25 TO 3.25 STEP .25
   LINE (-.08,Y)-(.12,Y)    'Zeichnet Einteilung der Y-
NEXT                        'Achse.
PSET (-9,3)                 'Stelle den ersten Punkt
                            'der Funktion dar.
FOR X = -9 TO 9 STEP .25
   Y = SQR(ABS(X))          'Das Argument von SQR darf
                            'nicht negativ sein
   LINE -(X,Y),2            'Zeichnet eine Gerade zum
NEXT                        'nächsten Punkt
```

STATIC-Anweisung

Funktion

Macht einfache Variablen oder Datenfelder lokal zu einer **DEF FN**-Funktion, **FUNCTION** oder **SUB** und erhält Werte zwischen Aufrufen.

Syntax

STATIC *Variablenliste*

Anmerkungen

Die *Variablenliste* einer **STATIC**-Anweisung hat folgende Syntax:

Variable [()][**AS** *Typ*] [, *Variable* [()][**AS** *Typ*]]...

Die *Variablenliste* hat folgende Argumente:

Argument	*Beschreibung*
Variable	Entweder ein Variablenname oder ein Datenfeldname.
AS *Typ*	Deklariert den Typ von *Variable*. Das Argument *Typ* kann **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, **STRING** oder ein benutzerdefinierter Typ sein.

Die **STATIC**-Anweisung kann nur in einer **SUB** oder **FUNCTION** oder **DEF FN**-Funktion erscheinen.

Frühere BASIC-Versionen erfordern die Anzahl der Dimensionen in Klammern nach einem Datenfeldnamen. In QuickBASIC ist die Anzahl der Dimensionen optional.

Variablen, die in einer **STATIC**-Anweisung deklariert werden, heben Variablen desselben Namens, die mit **DIM**- oder **COMMON**-Anweisungen im Modul-Ebenen-Code geteilt werden, auf. Variablen in einer **STATIC**-Anweisung heben auch globale Konstanten desselben Namens auf.

Normalerweise sind in **DEF FN**-Funktionen deklarierte Variablen global zu dem Modul; Sie können die **STATIC**-Anweisung innerhalb einer **DEF FN**-Anweisung jedoch dazu benutzen, eine Variable als lokal nur zu dieser Funktion zu deklarieren.

Hinweis Das **STATIC**-Attribut in **SUB**- und **FUNCTION**-Anweisungen deklariert die Standardeinstellung für Variablen als **STATIC**. Variablen, die denselben Namen haben wie Variablen, die mit dem Modul-Ebenen-Code geteilt werden, werden weiterhin geteilt. Im Unterschied dazu macht die **STATIC**-Anweisung spezielle Variablen zu **STATIC** und hebt alle Variablen auf, die mit dem Modul-Ebenen-Code geteilt werden. Der **$STATIC**-Metabefehl hat Auswirkungen auf die Zuweisung von Speicherplatz für Datenfelder. In Kapitel 2, "Datentypen", finden Sie weitere Informationen zu den Unterschieden zwischen diesen Anweisungen und Attributen.

Vergleichen Sie auch

COMMON, DEF FN, DIM, FUNCTION, SHARED, SUB

Beispiele

Dieses Beispiel stellt die Anweisungen **STATIC** und **SHARED** im Unterprogramm `ERHOEHEN` einander gegenüber. Die Variablen `W` und `Z` sind lokale Variablen dieses Unterprogramms, während die Variablen `WHL` und `ZAHL` von Hauptprogramm und Unterprogramm gemeinsam benutzt werden. Die Anweisung `STATIC W, Z` stellt sicher, daß `W` und `Z` jedesmal, wenn das Hauptprogramm das Unterprogramm aufruft, die letzten zugewiesenen Werte behalten.

```
Whl = 0 : Zahl = 0
PRINT "Vor der Schleife ist Whl = "; Whl;
PRINT ", Zahl = ";Zahl;
PRINT ", W = ";W;", Z = ";Z
FOR I = 1 TO 10
   CALL Erhoehen
NEXT
```

```
PRINT "Nach der Schleife ist Whl = ";Whl;
PRINT " Zahl =    ";Zahl;
PRINT ", W = ";W;", Z = ";Z
END

SUB Erhoehen STATIC
SHARED Whl,Zahl         'Gemeinsame Benutzung mit
                        'Hauptprogramm
STATIC W, Z             'Keine gemeinsame Benutzung mit
                        'Hauptprogramm
   W = W + 1            'W & Z zu Beginn gleich 0
   Z = Z + 2
   Whl = W
   Zahl = Z
END SUB
```

Ausgabe

```
Vor der Schleife ist Whl = 0 , Zahl = 0 , W = 0 , Z = 0
Nach der Schleife ist Whl = 10 , Zahl = 20, W = 0 , Z = 0
```

Das nachstehende Programm sucht in der angegebenen Datei nach jedem Vorkommen eines bestimmten Zeichenkettenausdrucks (gespeichert in der Variablen `Alt$`) und ersetzt diese Zeichenkette durch die in `Neu$` gespeicherte Zeichenkette. Der Name der so geänderten Datei ist der alte Dateiname mit der Erweiterung `.NEU`.

Außerdem gibt das Programm die Anzahl der Ersetzungen und der geänderten Zeilen aus:

```
INPUT "Dateiname";F1$
INPUT "Zu ersetzende Zeichenkette";Alt$
INPUT "Zu ersetzen durch";Neu$
Whl = 0 : Zahl = 0
M = LEN(Alt$)
OPEN F1$ FOR INPUT AS #1
CALL Erweiterung
OPEN F2$ FOR OUTPUT AS #2

DO WHILE NOT EOF(1)
   LINE INPUT #1, Temp$
   CALL Suche
   PRINT #2, Temp$
LOOP

CLOSE
PRINT "Es gab ";Whl;" Ersetzungen in ";Zahl;" Zeilen."
PRINT "Ersetzungen stehen in Datei ";F2$
END
```

```
SUB Erweiterung STATIC
SHARED F1$,F2$
   Zeichen = INSTR(F1$,".")
   IF Zeichen = 0 THEN
       F2$ = F1$ + ".NEU"
   ELSE
       F2$ = LEFT$(F1$,Zeichen - 1) + ".NEU"
   END IF
END SUB

SUB Suche STATIC
SHARED Temp$, Alt$, Neu$, Whl, Zahl, M
STATIC W
   Zeichen = INSTR(Temp$,Alt$)
   WHILE Zeichen
      Teil1$ = LEFT$(Temp$,Zeichen - 1)
      Teil2$ = MID$(Temp$,Zeichen + M)
      Temp$ = Teil1$ + Neu$ + Teil2$
      W = W + 1
      Zeichen = INSTR(Temp$,Alt$)
   WEND
   IF Whl = W THEN
      EXIT SUB
   ELSE
      Whl = W
      Zahl = Zahl + 1
   END IF
END SUB
```

Ausgabe

```
Dateiname? KAP1.S
Zu ersetzende Zeichenkette? KAPITEL 1
Zu ersetzen durch? EINLEITUNG
Es gab 23 Ersetzungen in 19 Zeilen.
Ersetzungen stehen in Datei KAP1.NEU
```

Die Datei `KAP1.NEU` enthält nun jede Zeile aus `KAP1.S`, wobei jedes Vorkommen der Zeichenkette `KAPITEL 1` durch `EINLEITUNG` ersetzt ist.

STICK-Funktion

Funktion

Gibt die x- und y-Koordinaten der beiden Joysticks an.

Syntax

STICK(*n*)

Anmerkungen

Das Argument ***n*** ist ein numerischer Ausdruck, dessen Wert eine vorzeichenlose ganze Zahl im Bereich von 0 bis 3 angibt:

Wert	*Zurückgegebener Wert*
0	Die x-Koordinate von Joystick A.
1	Die y-Koordinate von Joystick A, wenn `STICK (0)` zuletzt aufgerufen wurde.
2	Die x-Koordinate von Joystick B, wenn `STICK (0)` zuletzt aufgerufen wurde.
3	Die y-Koordinate von Joystick B, wenn `STICK (0)` zuletzt aufgerufen wurde.

Die x- und y-Koordinaten sind im Bereich von 1 bis 200.

Sie müssen **STICK (0)** verwenden, bevor Sie **STICK (1)**, **STICK (2)** oder **STICK** (3) benutzen. **STICK (0)** gibt nicht nur die x-Koordinate des Joysticks A zurück, speichert außerdem die Koordinaten des anderen Joysticks. Diese gespeicherten Koordinaten werden bei Aufruf von **STICK (1)** - **STICK (3)** zurückgegeben. Die folgenden Anweisungen geben zum Beispiel die Koordinaten vom Joystick B zurück:

```
TEMP = STICK (0)
PRINT STICK (2), STICK (3)
```

STOP-Anweisung

Funktion

Beendet das Programm.

Syntax

STOP

Anmerkungen

STOP-Anweisungen können an jeder Stelle eines Programmes benutzt werden, um die Ausführung zu beenden.

Läuft ein Programm in der QuickBASIC-Umgebung, läßt die **STOP**-Anweisung Dateien geöffnet und geht nicht zum Betriebssystem zurück. Im Unterschied hierzu schließt die **STOP**-Anweisung in einer selbständigen **.EXE**-Datei alle Dateien und kehrt zum Betriebssystem zurück.

Wenn Sie die **/d-**, **/e-** oder **/x**-Kompilieroptionen auf der **bc**-Befehlszeile benutzen, gibt die Anweisung **STOP** die Nummer der Zeile an, in der die Ausführung beendet wurde, wenn Ihr Programm Zeilennummern enthält. Wenn die **STOP**-Anweisung selbst keine Zeilennummer hat, wird die Nummer der zuletzt ausgeführten Zeile ausgegeben. Wenn Ihr Programm keine Zeilennummern enthält, wird als Zeilennummer 0 ausgegeben.

Früher wurde die **STOP**-Anweisung zum Debuggen benutzt. Die neuen Debug-Eigenschaften von QuickBASIC machen diese Verwendung von **STOP** unnötig.

STR$-Funktion

Funktion

Gibt eine Zeichenkettendarstellung des Wertes eines numerischen Ausdrucks an.

Syntax

STR$ (*Numerischer Ausdruck*)

Anmerkungen

Wenn *Numerischer Ausdruck* positiv ist, enthält die von **STR$** angegebene Zeichenkette ein führendes Leerzeichen.

Die Funktion **STR$** wird durch die Funktion **VAL** ergänzt.

Vergleichen Sie auch

VAL

Beispiel

Das nachstehende Beispiel enthält eine **FUNCTION**, die mit **STR$** eine Zahl in ihre Zeichenkettendarstellung umwandelt; anschließend entfernt die **FUNCTION** die führenden und nachgestellten Leerzeichen, die BASIC mit numerischer Ausgabe schreibt.

```
FUNCTION ZahlEntLeerz$(X) STATIC
   X$ = STR$(X)
   ZahlEntLeerz$ = LTRIM$(RTRIM$(X$))
END FUNCTION
PRINT "Geben Sie 0 ein zum Beenden."
DO
   INPUT "Berechne Kosinus von: ",Zahl
   IF Zahl = 0 THEN EXIT DO
   PRINT "COS(" ZahlEntLeerz$(Zahl) ") = "COS(Zahl)
LOOP
```

Ausgabe

```
Geben Sie 0 ein zum Beenden.
Berechne Kosinus von: 3.1
COS(3.1) = -.9991351
Berechne Kosinus von: 0
```

STRIG-Anweisung und -Funktion

Funktion

Gibt den Status eines Joystick-Knopfes an.

Syntax 1 (Anweisung)

STRIG {**ON** | **OFF**}

Syntax 2 (Funktion)

STRIG(*n*)

Anmerkungen

Die **STRIG**-Funktion wird benutzt, um den Status des Joystick-Knopfes zu überprüfen. In früheren BASIC-Versionen aktiviert **STRIG ON** die Überprüfung des Joystick-Knopfes; **STRIG OFF** schaltet die Überprüfung des Joystick-Knopfes aus. QuickBASIC ignoriert **STRIG ON**- und **STRIG OFF**-Anweisungen. Diese Anweisungen dienen nur der Kompatibilität zu älteren Versionen.

Der numerische Ausdruck *n* ist eine vorzeichenlose Ganzzahl im Bereich von 0 bis 7, der den zu testenden Knopf angibt. Die folgende Liste beschreibt die Werte, die von der **STRIG**(*n*)-Funktion bei verschiedenen Werten von *n* zurückgegeben werden.

Argument	*Zurückgegebener Wert*
0	-1, wenn der untere Knopf von Joystick A seit der letzten `STRIG (0)`-Anweisung gedrückt wurde; sonst 0.
1	-1, wenn der untere Knopf von Joystick A gerade gedrückt wird; sonst 0.
2	-1, wenn der untere Knopf von Joystick B seit der letzten `STRIG (2)`-Anweisung gedrückt wurde; sonst 0.
3	-1, wenn der untere Knopf von Joystick B gerade gedrückt wird; sonst 0.
4	-1, wenn der obere Knopf von Joystick A seit der letzten `STRIG (4)`-Anweisung gedrückt wurde; sonst 0.
5	-1, wenn der obere Knopf von Joystick A gerade gedrückt wird; sonst 0.
6	-1, wenn der obere Knopf von Joystick B seit der letzten `STRIG (6)`-Anweisung gedrückt wurde; sonst 0.
7	-1, wenn der obere Knopf von Joystick B gerade gedrückt wird; sonst 0.

Sie können auch die Ereignisverfolgung benutzen, um Informationen vom Joystick durch die Verwendung der **ON STRIG**-Anweisung zu erhalten (siehe **ON** *Ereignis*-Abschnitt). Sie können die **STRIG**-Funktion nicht innerhalb einer Joystick-Ereignisverfolgung benutzen, da das Ereignis, das die Verfolgung ausgelöst hat, zerstört ist.

Unterschied zu BASICA

Wenn Sie von der **bc**-Befehlszeile aus kompilieren, müssen Sie die **/v**- oder **/w**-Option verwenden, wenn ein Programm eine **STRIG**-Anweisung enthält.

Vergleichen Sie auch

ON *Ereignis*, **STRIG ON, STRIG OFF, STRIG STOP**

Beispiel

```
'Warte, bis Knopf A gedrückt wird:
DO
  WurdeABet = STRIG(0)
LOOP UNTIL WurdeABet
'Solange Knopf A gedrückt wird, Tonsignal:
DO
  WurdeABet = STRIG(1)
   BEEP
LOOP WHILE WurdeABet
```

STRIG ON-, OFF-, und STOP-Anweisungen

Funktion

Aktiviert, deaktiviert oder sperrt die Ereignisverfolgung für den Joystick.

Syntax

STRIG(*n***)ON**

STRIG(*n***)OFF**

STRIG(*n***)STOP**

Anmerkungen

Das Argument *n* ist ein numerischer Ausdruck, der den zu verfolgenden Joystick-Knopf angibt:

Wert	***Knopf***
0	Unterer Knopf, Joystick A
2	Unterer Knopf, Joystick B
4	Oberer Knopf, Joystick A
6	Oberer Knopf, Joystick B

Die Anweisung **STRIG(*n*) ON** aktiviert die Ereignisverfolgung für den Joystick durch eine **ON STRIG**-Anweisung (siehe hierzu die Nachschlageseiten zu **ON** *Ereignis*). Während die Verfolgung aktiviert ist und die Anweisung **ON STRIG** eine Zeilennummer ungleich Null enthält, prüft BASIC zwischen jeder Anweisung, ob der Joystick-Knopf gedrückt wurde.

Die Anweisung **STRIG(*n*) OFF** deaktiviert die Ereignisverfolgung. Wenn ein nachfolgendes Ereignis eintritt (d. h., wenn der Knopf gedrückt wird), ist es bei der Ausführung des nächsten **STRIG ON** nicht gespeichert.

Die Anweisung **STRIG(*n*) STOP** sperrt die Ereignisverfolgung. Wenn ein Ereignis eintritt, wird dieses vermerkt, und die Ereignisverfolgung erfolgt, sobald die Erfassung erneut aktiviert ist.

Vergleichen Sie auch

ON *Ereignis*, **STRIG**

STRING$-Funktion

Funktion

Gibt eine Zeichenkette an, deren Zeichen alle einen gegebenen ASCII-Code haben, oder deren Zeichen alle das erste Zeichen eines Zeichenkettenausdrucks sind.

Syntax

STRING$(*m*, *n*)
STRING$(*m*, *Zeichenkettenausdruck*)

Anmerkungen

Die Funktion **STRING$** hat die folgenden Argumente:

Argument	*Beschreibung*
m	Ein numerischer Ausdruck, der die Länge der anzugebenden Zeichenkette anzeigt.
n	Der ASCII-Code des für die Bildung der Zeichenkette verwendeten Zeichens. Es ist ein numerischer Ausdruck, der einen ganzzahligen Wert im Bereich von 0-255 ermittelt.
Zeichenketten-ausdruck	Der Zeichenkettenausdruck, dessen erstes Zeichen zur Bildung der anzugebenden Zeichenkette verwendet wird.

Beispiele

Das folgende Beispiel benutzt **STRING$**, um einen Teil eines Berichtskopfes zu erzeugen:

```
Strich$ = STRING$(10,45)
PRINT Strich$;"MONATLICHER BERICHT";Strich$
```

Ausgabe

```
----------MONATLICHER BERICHT----------
```

Das nachstehende Programm verwendet **STRING$** zur Erstellung eines Balkendiagramms:

```
PRINT TAB(7);"Tagesdurchschnittstemperaturen in Seattle"
PRINT
FOR Monat = 1 TO 12 STEP 2
    READ Monat$, Temp
    'Gib Temp - 35 Sterne aus.
    PRINT Monat$;" +"; STRING$(Temp-35,"*")
    PRINT "    I"
NEXT Monat
PRINT "    +";
FOR X = 1 TO 7
    PRINT "----+";
NEXT X
PRINT
FOR X = 4 to 39 STEP 5
    PRINT TAB(X); X+31;
NEXT X
PRINT
DATA Jan, 40, Mrz, 46, Mai, 56
DATA Jul, 66, Sep, 61, Nov, 46
```

Ausgabe

```
Tagesdurchschnittstemperaturen in Seattle
Jan +*****
    I
Mrz +***********
    I
Mai +*********************
    I
Jul +*******************************
    I
Sep +**************************
    I
Nov +***********
    I
    +----+----+----+----+----+----+----+
    35   40   45   50   55   60   65   70
```

SUB-Anweisungen

Funktion

Kennzeichnet Anfang und Ende eines Unterprogramms.

Syntax

SUB *Globalname*[(*Parameterliste*)][**STATIC**]
.
.
.
[**EXIT SUB**]
.
.
.
END SUB

Anmerkungen

Die Anweisung **SUB** hat die folgenden Argumente:

Argument	*Beschreibung*
Globalname	Ein Variablenname von bis zu 40 Zeichen Länge. Dieser Name darf in keiner anderen **SUB**- oder **FUNCTION**-Anweisung im selben Programm oder in der Benutzerbibliothek erscheinen.
Parameterliste	Enthält die Namen von einfachen Variablen und Datenfeldern, die in das Unterprogramm übergeben werden, wenn die **SUB** aufgerufen wird. Jeder Name wird vom vorhergehenden Namen durch ein Komma getrennt. Beachten Sie bitte, daß diese Variablen und Datenfelder als Referenz übergeben werden, so daß jede Änderung eines Argumentwertes im Unterprogramm dessen Wert auch im aufrufenden Programm verändert. Eine umfassende Beschreibung der Syntax finden Sie weiter unten.

Eine **SUB**-*Parameterliste* hat folgende Syntax:

Variable[( )][**AS** *Typ*][,*Variable*[( )][**AS** *Typ*]]

Eine *Variable* ist ein BASIC-Variablenname. Frühere BASIC-Versionen erforderten die Anzahl der Dimensionen in Klammern nach dem Datenfeldnamen. In QuickBASIC ist die Anzahl der Dimensionen nicht erforderlich. Das Argument *Typ* ist der Typ der Variablen. Das *Typ*-Argument kann **INTEGER, LONG, SINGLE, DOUBLE, STRING** oder benutzerdefinierten Typs sein.

Ein Unterprogramm ist eine separate Prozedur, wie eine **FUNCTION**. Dennoch kann eine **SUB** im Unterschied zu einer **FUNCTION** nicht in einem Ausdruck verwendet werden.

SUB und **END SUB** markieren den Anfang und das Ende eines Unterprogramms. Sie können auch die wahlfreie **EXIT SUB**-Anweisung benutzen, um ein Unterprogramm zu verlassen.

Unterprogramme werden mit der Anweisung **CALL** aufgerufen, oder durch den Namen des Unterprogramms, gefolgt von der Argumentenliste. Schlagen Sie hierzu unter der Beschreibung der **CALL**-Anweisung nach.

QuickBASIC-Unterprogramme können rekursiv sein - sie können sich selbst aufrufen, um eine bestimmte Aufgabe zu erfüllen. Siehe zweites Beispiel unten und Abschnitt 4.4, "Rekursionen", in diesem Buch.

Das Attribut **STATIC** zeigt an, daß alle Variablen, die lokal zu der **SUB** sind, **STATIC** sind. Ihre Werte bleiben zwischen den Aufrufen erhalten. Die Verwendung das **STATIC** Schlüsselwortes erhöht die Ausführungsgeschwindigkeit etwas. **STATIC** wird normalerweise nicht in rekursiven Unterprogrammen verwendet. Siehe hierzu die Beispiele unten.

Alle Unterprogrammvariablen oder -datenfelder werden als lokal zu diesem Unterprogramm betrachtet, außer wenn sie in einer **SHARED**-Anweisung explizit als gemeinsam benutzte Variablen deklariert werden.

Sie können keine **SUB**-Prozeduren, **DEF FN**-Funktionen oder **FUNCTION**-Prozeduren innerhalb einer **SUB**-Prozedur definieren.

Hinweis Sie können Unterprogramme nicht mit **GOSUB, GOTO** oder **RETURN** aufrufen oder beenden.

Vergleichen Sie auch

CALL (BASIC); DECLARE (BASIC); SHARED; STATIC; Kapitel 2, "Prozeduren: Unterprogramme und Funktionen", in *Programmieren in BASIC: Ausgewählte Themen*

Beispiele

In diesem Beispiel ruft das Hauptprogramm das Unterprogramm `ZEILSUCH` auf, das in jeder Zeile der Eingabe aus Datei `D$` nach der angegebenen Zeichenkette `M$` sucht. Wenn das Unterprogramm `M$` in einer Zeile findet, gibt es diese Zeile zusammen mit ihrer Zeilennummer aus. Beachten Sie, daß der Wert von `Zahl` zwischen den Aufrufen erhalten bleibt, da das **STATIC**-Schlüsselwort in der **SUB**-Anweisung benutzt wird.

```
INPUT "Zu durchsuchende Datei";D$
INPUT "Zu suchendes Muster";M$
OPEN D$ FOR INPUT AS #1
DO WHILE NOT EOF(1)
   LINE INPUT #1, Test$
   CALL Zeilsuch(Test$,M$)
LOOP
SUB Zeilsuch(Test$,M$) STATIC
   Zahl = Zahl + 1
   X = INSTR(Test$,M$)
   IF X > 0 THEN PRINT "Zeile Nr.";Zahl;": ";Test$
END SUB
```

Ausgabe

```
Zu durchsuchende Datei? such.bas
Zu suchendes Muster? SUB
Zeile Nr. 9 : SUB Zeilsuch(Test$,M$) STATIC
Zeile Nr. 13 : END SUB
```

Dieses Beispiel benutzt eine rekursive **SUB**, um das klassische Problem "Die Türme von Hanoi" für eine variable Anzahl von Scheiben zu lösen. Bei einer großen Anzahl von Scheiben muß die Stapelgröße mit der **CLEAR**-Anweisung erhöht werden.

```
DECLARE SUB Hanoi (N%, Quel%, Ziel%, Zwisch%)
DEFINT A-Z
DIM SHARED BewZaehl
BewZaehl = 0
CLS
PRINT "Die Türme von Hanoi"
INPUT "Geben Sie die Anzahl der Scheiben ein:"; _
       Scheiben
'Rufe die rekursive Hanoi-SUB auf, um das Problem zu
'lösen.
CALL Hanoi(Scheiben, 1, 3, 2)
```

```
PRINT "Problem komplett in";BewZaehl; "Schritten."
END
'Bewege N Scheiben von Quel zu Ziel; verwende dabei
'Zwisch als Ablage
SUB Hanoi (N, Quel, Ziel, Zwisch)
   IF N <= 1 THEN
      '1 Scheibe. Bewege sie direkt zu Ziel.
      PRINT "Bewege eine Scheibe von"; Quel;
      PRINT "zu"; Ziel
      BewZaehl = BewZaehl + 1
   ELSE
      'Mehr als eine Scheibe.
      'Bewege die N-1 Scheiben zu Zwisch, verwende
      'dabei Ziel als Ablage.
      CALL Hanoi(N - 1, Quel, Zwisch, Ziel)
      'Bewege die N-te Scheibe direkt zu Ziel.
      PRINT "Bewege eine Scheibe von"; Quel;
      PRINT "zu"; Ziel:
      BewZaehl = BewZaehl + 1
      'Bewege die N-1 Scheiben zu Ziel, verwende dabei
      'Quel als Ablage.
      CALL Hanoi(N-1, Zwisch, Ziel, Quel)
   END IF
END SUB
```

Ausgabe

```
Die Türme von Hanoi
Geben Sie die Anzahl der Scheiben ein: 3
Bewege eine Scheibe von 1 zu 3
Bewege eine Scheibe von 1 zu 2
Bewege eine Scheibe von 3 zu 2
Bewege eine Scheibe von 1 zu 3
Bewege eine Scheibe von 2 zu 1
Bewege eine Scheibe von 2 zu 3
Bewege eine Scheibe von 1 zu 3
Problem komplett in 7 Schritten.
```

SWAP-Anweisung

Funktion

Tauscht die Werte von zwei Variablen aus.

Syntax

SWAP *Variable1,Variable2*

Anmerkungen

Jeder Variablentyp (Ganzzahl, lange Ganzzahl, Variable einfacher Genauigkeit, Variable doppelter Genauigkeit, Zeichenkette oder Verbundvariable) kann getauscht werden. Die beiden Variablen müssen jedoch exakt vom gleichen Typ sein, anderenfalls erscheint eine Fehlermeldung (`Unverträgliche Datentypen`). Zum Beispiel wird der Versuch, eine Ganzzahl mit einem Wert einfacher Genauigkeit zu tauschen, `Unverträgliche Datentypen` hervorrufen.

Beispiel

Die folgende Unterroutine (`ShellSort`) sortiert die Elemente eines Zeichenketten-Datenfeldes in absteigender Reihenfolge unter Verwendung eines Shellsorts. `ShellSort` benutzt **SWAP**, um Datenfeldelemente auszutauschen, die nicht in richtiger Reihenfolge sind.

```
'Sortiere die Liste der Wörter unter Verwendung eines
'Shellsorts.
SUB ShellSort (Feld$(), Num%) STATIC
   Span% = Num% \ 2
   DO WHILE Span% > 0
      FOR I% = Span% TO Num% - 1
         J% = I% - Span% + 1
         FOR J% = (I% - Span% + 1) TO 1 STEP -Span%
              IF Feld$(J%) <= Feld$(J% + Span%) THEN _
                 EXIT FOR
              'Tausche Datenfeldelemente, die nicht in
              'Reihenfolge sind, aus.
              SWAP Feld$(J%), Feld$(J% + Span%)
         NEXT J%
```

```
        NEXT I%
        Span% = Span% \ 2
    LOOP
END SUB
```

SYSTEM-Anweisung

Funktion

Schließt alle geöffneten Dateien und gibt die Kontrolle an das Betriebssystem zurück.

Syntax

SYSTEM

Anmerkungen

Wenn ein **SYSTEM**-Befehl ausgeführt wird, werden alle Dateien geschlossen, und BASIC kehrt zum Betriebssystem zurück (bei selbständig ausführbaren Programmen) oder beendet die Programmausführung (wenn sich das Programm in der QuickBASIC-Umgebung befindet).

Hinweis Ein Programm, das eine **SYSTEM**-Anweisung enthält, kehrt zum Betriebssystem zurück, wenn das Programm von der QuickBASIC-Befehlszeile mit der Option **/run** gestartet wird.

Wird die **SYSTEM**-Anweisung im Befehlsfenster eingegeben, hält QuickBASIC an.

Unterschied zu BASICA

END und **SYSTEM** sind in BASICA verschieden, haben in QuickBASIC aber die gleiche Wirkung.

TAB-Funktion

Funktion

Bewegt die Ausgabeposition.

Syntax

TAB(*Spalte*)

Anmerkungen

Das Argument *Spalte* ist ein numerischer Ausdruck, der die Spaltennummer der neuen Ausgabeposition darstellt. Wenn sich die aktuelle Ausgabeposition bereits hinter *Spalte* befindet, bewegt die Funktion **TAB** die Ausgabeposition in diese Spalte auf der nächsten Zeile. Spalte 1 ist die am weitesten links liegende Position. Die am weitesten rechts liegende Position ist die aktuelle Zeilenbreite des Ausgabegerätes minus 1. Falls *Spalte* größer als diese Ausgabebreite ist, wird die **TAB**-Position neu berechnet (die Ausgabeposition wird *Spalte* **MOD** *Breite*). Ist *Spalte* kleiner als 1, so verschiebt **TAB** die Ausgabeposition nach Spalte 1.

TAB kann nur in **PRINT**- und **LPRINT**-Anweisungen verwendet werden.

Beispiele

Das folgende Beispiel benutzt **TAB**, um die Ausgabespalten festzulegen:

```
FOR I = 1 TO 4
   READ A$,B$
   PRINT A$ TAB(25) B$
NEXT
DATA NAME, BETRAG,,,G.T. JONAS, DM 25.00, _
     H.L.STEFFENS, DM 32.25
```

Ausgabe

```
NAME                       BETRAG
G.T. JONAS                 DM 25.00
H.L. STEFFENS              DM 32.25
```

Das folgende Beispiel zeigt die Effekte unterschiedlicher Werte für das **TAB**-Argument:

```
'Angenommene Bildschirmbreite sei 80 Spalten
PRINT TAB(1287) "EINS"
PRINT TAB(255) "ZWEI"
PRINT TAB(-5) "DREI"
PRINT "123456789012345678901234567890" TAB(20) "VIER"
```

Ausgabe

```

      EINS
              ZWEI
DREI
123456789012345678901234567890
                   VIER
```

TAN-Funktion

Funktion

Gibt den Tangens von *x* an, wobei *x* im Bogenmaß steht.

Syntax

TAN(*x*)

Anmerkungen

TAN wird mit einfacher Genauigkeit berechnet. Falls das Argument *x* jedoch ein Wert doppelter Genauigkeit ist, wird **TAN** ebenfalls mit doppelter Genauigkeit berechnet.

Unterschied zu BASICA

Wenn die Berechnung von **TAN** in BASICA einen Überlauf zur Folge hat, zeigt der Interpreter die Fehlermeldung `Überlauf` an, das Ergebnis wird Unendlich (Maschinenabhängig) und die Ausführung wird fortgesetzt.

Läuft **TAN** in QuickBASIC über, so wird kein Unendlich (der Maschine) angezeigt und die Ausführung gestoppt (es sei denn, das Programm enthält eine Fehlerbehandlungsroutine).

Beispiel

Das nachstehende Beispiel berechnet die Höhe eines Objektes unter Verwendung der Entfernung und des Höhenwinkels. Das Programm zeichnet das durch die Basis und die berechnete Höhe erzeugte Dreieck.

```
SCREEN 2
INPUT "LÄNGE DER BASIS: ", BasLaeng
INPUT "HÖHENWINKEL (GRAD, MINUTEN): ",Grad, Min
Winkel = (3.141593/180)*(Grad + Min/60) 'Wandelt in
                                        'Bogenmaß um
Hoehe = BasLaeng*TAN(Winkel)  'Berechnet Höhe
PRINT "HÖHE =" Hoehe
H = 180 - Hoehe
B = 15 + BasLaeng
LINE (15,180)-(B,180)         'Zeichnet Dreieck
LINE -(B,H)
LINE -(10,180)
LOCATE 24,1 : PRINT "Beliebige Taste betätigen...";
DO
LOOP WHILE INKEY$=""
```

TIME$-Anweisung

Funktion

Setzt die Zeit.

Syntax

TIME$ = *Zeichenkettenausdruck*

Anmerkungen

Der *Zeichenkettenausdruck* muß eines der folgenden Formate haben:

Format	***Beschreibung***
hh	Setzt die Stunde; Standardeinstellung für Minuten und Sekunden ist 00.
hh:mm	Setzt die Stunde und die Minute; Standardeinstellung für Sekunden ist 00.
hh:mm:ss	Setzt die Stunde, Minute und Sekunde.

Es wird eine 24-Stunden-Uhr verwendet; geben Sie 8.00 Uhr abends daher als 20:00:00 ein.

Diese Anweisung ergänzt die Funktion **TIME$**, die die aktuelle Zeit ermittelt.

Vergleichen Sie auch

TIME$-Funktion

Beispiel

Das folgende Beispiel setzt die aktuelle Zeit auf 8.00 Uhr:

```
TIME$="08:00:00"
```

TIME$-Funktion

Funktion

Gibt die aktuelle Uhrzeit des Betriebssystems an.

Syntax

TIME$

Anmerkungen

Die Funktion **TIME$** gibt eine Zeichenkette aus acht Zeichen im Format *hh:mm:ss* an. Dabei steht *hh* für Stunden (00 - 23), *mm* für die Minuten (00 - 59) und *ss* für die Sekunden (00 - 59). Verwendet wird eine 24-Stunden-Uhr; 8.00 Uhr abends wird daher als 20:00:00 angezeigt.

Zum Setzen der Zeit verwenden Sie die Anweisung **TIME$**.

Vergleichen Sie auch

TIME$-Anweisung, TIMER

Beispiel

Das folgende Beispiel wandelt die von **TIME$** zurückgegebene 24-Stunden-Zeit in eine 12-Stunden-Zeit um:

```
'Wandelt die von TIME$ verwendete 24-Stunden-Zeit in
'die im englischen Sprachraum verwendete 12-Stunden-
'Zeit mit "AM" bzw. "PM" um:
T$ = TIME$
Hr = VAL(T$)
IF Hr < 12 THEN Ampm$ = " AM" ELSE Ampm$ = " PM"
IF Hr > 12 THEN Hr = Hr - 12
PRINT "Uhrzeit: " STR$(Hr) RIGHT$(T$,6) Ampm$
```

Ausgabe

```
Uhrzeit: 11:26:31 AM
```

TIMER-Funktion

Funktion

Gibt die Anzahl der seit 0.00 Uhr vergangenen Sekunden an.

Syntax

TIMER

Anmerkungen

Die Funktion **TIMER** kann mit der Anweisung **RANDOMIZE** zur Generierung einer Zufallszahl verwendet werden. Außerdem kann sie dazu dienen, Laufzeiten von Programmen oder Programmteilen zu stoppen.

Beispiel

Das nachstehende Programm sucht nach der Anzahl der Primzahlen von 3 bis 10.000 unter Verwendung einer Variation des "Sieb von Eratosthenes". Dabei wird die Funktion **TIMER** verwendet, um die Laufzeit des Programms zu stoppen.

```
DEFINT A-Z
CONST UNMARK = 0, MARKIT = NOT UNMARK
DIM Mark(10000)
Start! = TIMER
Num = 0
FOR N = 3 TO 10000 STEP 2
   IF NOT Mark(N) THEN
      'PRINT N,  'Zur Ausgabe der Primzahlen muß das
                 'Kommentarzeichen vor der PRINT-
                 'Anweisung entfernt werden
      Delta = 2*N
      FOR I = 3*N TO 10000 STEP Delta
          Mark(I) = MARKIT
      NEXT
      Num = Num + 1
   END IF
NEXT
Ende! = TIMER
PRINT : PRINT "Das Programm benötigte " Ende!-Start!;
PRINT "Sekunden, um die" Num "Primzahlen zu finden"
END
```

Ausgabe

```
Das Programm benötigte .1601563 Sekunden, um die 1228
Primzahlen zu finden.
```

TIMER ON-, OFF-, und STOP-Anweisungen

Funktion

Aktiviert, deaktiviert oder sperrt die Ereignisverfolgung des Zeitgebers.

Syntax

TIMER ON

TIMER OFF

TIMER STOP

Anmerkungen

TIMER ON aktiviert die Zeitgeber-Ereignisverfolgung durch eine **ON TIMER**-Anweisung (siehe **ON** *Ereignis*-Anweisung). Während die Ereignisverfolgung aktiviert ist, wird nach jeder Anweisung geprüft, ob die angegebene Zeit vergangen ist. Ist dies der Fall, wird die **ON TIMER**-Ereignisbehandlungsroutine ausgeführt.

TIMER OFF deaktiviert die Zeitgeber-Ereignisverfolgung. Tritt ein Ereignis ein, wird es nicht gespeichert, wenn eine nachfolgende Anweisung **TIMER ON** verwendet wird.

TIMER STOP deaktiviert die Zeitgeber-Ereignisverfolgung; falls jedoch ein Ereignis eintritt, wird es gespeichert und die **ON TIMER**-Ereignisbehandlungsroutine wird ausgeführt, sobald die Verfolgung aktiviert wird.

Vergleichen Sie auch

ON *Ereignis*

Beispiel

Dieses Beispiel zeigt in Zeile 1 die Tageszeit an und aktualisiert die Anzeige einmal pro Minute.

```
TIMER ON
ON TIMER(60) GOSUB Anzeige
DO WHILE INKEY$ = "" : LOOP
END

Anzeige:
   AltZeile = CRSLIN  'Speichert aktuelle Zeile
   AltSpalte = POS(0) 'Speichert aktuelle Spalte
   LOCATE 1,1 : PRINT TIME$;
   LOCATE AltZeile,AltSpalte  'Zeile und Spalte auf
                              'alte Position
RETURN
```

TRON-, TROFF-Anweisungen

Funktion

Verfolgt die Ausführung von Programmanweisungen.

Syntax

TRON

TROFF

Anmerkungen

In der QuickBASIC-Umgebung hat die Ausführung der **TRON**-Anweisung die gleiche Auswirkung wie die Auswahl von **Verfolgen Ein** aus dem Menü **Debug** – jede Anweisung wird bei ihrer Ausführung auf dem Bildschirm hervorgehoben.

Die Anweisung **TROFF** schaltet die Programmverfolgung aus.

Die Anweisungen **TRON** und **TROFF** zeigen Zeilennummern nur an, wenn mit der Option Debug oder auf der **bc**-Befehlszeile mit der Option **/d** kompiliert wurde.

Hinweis Durch die Debug-Eigenschaften der QuickBASIC-Umgebung werden diese Anweisungen überflüssig.

TYPE-Anweisung

Funktion

Definiert einen Datentyp, der ein oder mehrere Element/e enthält.

Syntax

TYPE *Benutzertyp*
  Elementname **AS** *Typname*
  Elementname **AS** *Typname*
.
.
.
END TYPE

Anmerkungen

Die Anweisung **TYPE** hat folgende Argumente:

Argument	*Beschreibung*
Benutzertyp	Ein dem benutzerdefinierten Datentyp zugeordneter Name. Folgt den gleichen Regeln wie für BASIC-Variablennamen.
Elementname	Der Name eines Elements des benutzerdefinierten Datentyps. Folgt den gleichen Regeln wie für BASIC-Variablennamen. Kann nicht der Name eines Datenfeldes sein.
Typname	Jeder der folgenden BASIC-Datentypen: **INTEGER**, **LONG**, **SINGLE**, **DOUBLE**, Zeichenkette fester Länge (siehe unten) oder benutzerdefinierter Typ.

Hinweis Zeichenketten in Benutzertypen müssen Zeichenketten fester Länge sein. Zeichenkettenlängen werden durch einen Stern und eine numerische Konstante angegeben. Die folgende Zeile definiert zum Beispiel ein Element namens `SchluessWort` in einem benutzerdefinierten Typ als eine Zeichenkette mit der Länge 40:

```
SchluessWort AS STRING*40
```

Bevor ein benutzerdefinierter Typ in einem Programm verwendet werden kann, muß er in einer **TYPE**-Deklaration deklariert worden sein. Obwohl ein benutzerdefinierter Typ nur im Modul-Ebenen-Code deklariert werden kann, können Sie eine Variable an beliebiger Stelle in dem Modul, sogar in einer **SUB** oder **FUNCTION**, als benutzerdefinierten Typ angeben.

Verwenden Sie **DIM-**, **REDIM-**, **COMMON-**, **STATIC-** oder **SHARED**-Anweisungen, um eine Variable als benutzerdefinierten Typ zu deklarieren.

Beispiel

Das folgende Beispiel simuliert unter Verwendung eines benutzerdefinierten Typs ein Kartenspiel. Das Programm bildet ein Kartenspiel (ein Datenfeld vom benutzerdefinierten Typ `Karte`), mischt das Kartenspiel und zeigt die ersten fünf Karten an.

```
' Dieses Programm verwendet einen benutzerdefinierten Typ,
' um ein Kartenspiel zu simulieren.
' Definiere den Typ Karte--einen Ganzzahlwert und eine
' Zeichenkette, die die Farbe angibt.
TYPE Karte
   Wert AS INTEGER
   Farbe AS STRING*9
END TYPE
DEFINT A-Z
' Definiere das Kartenspiel als ein Datenfeld mit 52
' Elementen für die Karten.
DIM Blatt(1 TO 52) AS Karte
' Bilde, mische und zeige die ersten fünf Karten.
CALL BildeBlatt(Blatt())
CALL Mische(Blatt())
FOR I% = 1 TO 5
    CALL ZeigeKarte(Blatt(I%))
NEXT I%
' Bilde das Blatt--fülle das Datenfeld der Karten mit
' passenden Werten.
SUB BildeBlatt(Blatt(1) AS Karte) STATIC
DIM Farben(4) AS STRING*9
   Farben(1) = "Karo"
   Farben(2) = "Herz"
   Farben(3) = "Pik"
   Farben(4) = "Kreuz"
' Diese Schleife steuert das Blatt.
   FOR I% = 1 TO 4
    ' Diese Schleife steuert den Bildwert.
      FOR J% = 1 TO 13
       ' Lege fest, welche Karte (1...52) wir
       ' erzeugen.
         KarteZahl% = J% + (I% - 1)*13
       ' Bringe den Bildwert und die Farbe in die
       ' Karte.
         Blatt(KarteZahl%).Wert = J%
         Blatt(KarteZahl%).Farbe = Farben(I%)
      NEXT J%
   NEXT I%
END SUB
' Mische das Kartenspiel (ein Datenfeld, das
' Kartenelemente
' enthält).
SUB Mische(Blatt(1) AS Karte) STATIC
```

```
    RANDOMIZE TIMER
' Mische durch Austauschen von 1000 zufällig
' ausgewählten Kartenpaaren.
    FOR I% = 1 TO 1000
        KarteEins% = INT(52 * RND + 1)
        KarteZwei% = INT(52 * RND + 1)
     ' Beachten Sie, daß SWAP mit Datenfeldern von
     ' Benutzertypen arbeitet.
        SWAP Blatt(KarteEins%),Blatt(KarteZwei%)
    NEXT I%
END SUB
' Zeige eine einzelne Karte an durch Übertragen und
' Schreiben des Bildwertes und der Farbe.
SUB ZeigeKarte (EinzelKarte AS Karte) STATIC
    SELECT CASE EinzelKarte.Wert
       CASE 13
          PRINT "König";
       CASE 12
          PRINT "Dame ";
       CASE 11
          PRINT "Bube ";
       CASE  1
          PRINT "As   ";
       CASE ELSE
          PRINT USING "  ## ";EinzelKarte.Wert;
    END SELECT
    PRINT " ";EinzelKarte.Farbe
END SUB
```

Ausgabe

```
   3  Herz
   4  Pik
   3  Kreuz
Bube  Herz
   8  Karo
```

UBOUND-Funktion

Funktion

Ermittelt die obere Grenze (den größten vorhandenen Index) für die angegebene Dimension eines Datenfeldes.

Syntax

UBOUND (*Datenfeld*[,*Dimension*])

Anmerkungen

Das Argument *Dimension* ist eine Genzzahl von 1 bis zur Anzahl der Dimensionen in *Datenfeld.*

Für ein wie folgt dimensioniertes Datenfeld gibt **UBOUND** die nachfolgend aufgelisteten Werte aus:

```
DIM A(1 TO 100, 1 TO 50, -3 TO 4)
```

Aufruf	*Zurückgegebener Wert*
`UBOUND (A,1)`	`100`
`UBOUND (A,2)`	`50`
`UBOUND (A,3)`	`4`

Für eindimensionale Datenfelder können Sie die verkürzte Syntax **UBOUND**(*Datenfeld*) verwenden, weil der Standardwert für *Dimension* 1 ist.

Zur Ermittlung der unteren Grenze einer Datenfelddimension dient die Funktion **LBOUND.**

Vergleichen Sie auch

LBOUND

Beispiel

LBOUND und **UBOUND** können zusammen verwendet werden, um die Größe eines an ein Unterprogramm übergebenen Datenfeldes zu ermitteln, wie im folgenden Programmausschnitt gezeigt:

```
CALL DRUCKMAT(DATENFELD())
.
.
.
SUB DRUCKMAT(A()) STATIC
   FOR I% = LBOUND(A,1) TO UBOUND(A,1)
       FOR J% = LBOUND(A,2) TO UBOUND(A,2)
           PRINT A(I%,J%);" ";
       NEXT J%
   PRINT:PRINT
   NEXT I%
END SUB
```

UCASE$-Funktion

Funktion

Gibt einen Zeichenkettenausdruck an, in dem alle Buchstaben groß geschrieben sind.

Syntax

UCASE$ (*Zeichenkettenausdruck*)

Anmerkungen

Das Argument *Zeichenkettenausdruck* kann jeder Zeichenkettenausdruck sein.

Die Funktion **UCASE$** arbeitet sowohl mit Zeichenketten variabler als auch fester Länge.

Die Anweisungen **UCASE$** und **LCASE$** sind hilfreich, um Zeichenkettenvergleiche unabhängig von Groß-/Kleinschreibung vorzunehmen.

Vergleichen Sie auch

LCASE$

Beispiel

Das folgende Programm enthält eine **FUNCTION**, `JaFrage`, die abhängig von der Benutzerantwort einen Booleschen Wert angibt. Die Funktion `JaFrage` verwendet **UCASE$**, um eine von Groß-/Kleinschreibung unabhängige Prüfung der Benutzerantwort durchzuführen.

```
DEFINT A-Z
FUNCTION JaFrage(Anfrage$,Zeile,Spalte) STATIC
   AltZeile=CRSLIN
   AltSpalte=POS(0)
   'Schreib Anfrage in Zeile, Spalte
   LOCATE Zeile,Spalte : PRINT Anfrage$ "(J/N):";
   DO
        'Veranlasse den Benutzer, eine Taste zu
        'betätigen.
        DO
            Antw$=INKEY$
        LOOP WHILE Antw$=""
        Antw$=UCASE$(Antw$)
        'Stelle fest, ob ja oder nein...
        IF Antw$="J" OR Antw$="N" THEN
           EXIT DO
        ELSE
             BEEP
        END IF
   LOOP
   'Gib die Antwort auf der Zeile aus.
   PRINT Antw$;
   'Bringe den Cursor zurück an die alte Position
   LOCATE AltZeile,AltSpalte
   'Gib einen Booleschen Wert als Ergebnis des Tests
   'aus.
   JaFrage=(Antw$="J")
END FUNCTION
DO
LOOP WHILE NOT JaFrage("Kennen Sie München?",12,5)
```

UNLOCK-Anweisung

Funktion

Hebt die für Teile einer Datei geltenden Zugriffsbeschränkungen auf.

Syntax

UNLOCK [#]*Dateinummer*[,{*Satz* | [*Beginn*]**TO** *Ende*}]

Anmerkungen

Die **UNLOCK**-Anweisung wird nur nach einer **LOCK**-Anweisung verwendet. Ausführliche Informationen und Beispiele hierzu finden Sie auf den Nachschlageseiten der **LOCK**-Anweisung.

Für Dateien im Binärmodus repräsentieren die Argumente *Satz*, *Beginn* und *Ende* die Anzahl der Bytes relativ zum Beginn der Datei. Das erste Byte in einer Datei ist Byte 1.

Für Direktzugriffs-Dateien stehen diese Argumente für die Nummer eines Satzes relativ zum Beginn der Datei. Der erste Satz ist der Satz 1.

Vergleichen Sie auch

LOCK

VAL-Funktion

Funktion

Gibt den numerischen Wert einer Zeichenkette von Ziffern an.

Syntax

VAL (*Zeichenkettenausdruck*)

Anmerkungen

Der *Zeichenkettenausdruck* ist eine Folge von Zeichen, die als ein numerischer Wert interpretiert werden kann. Die Funktion **VAL** beendet das Lesen der Zeichenkette beim ersten Zeichen, das sie nicht als Teil einer Zahl erkennen kann. Außerdem entfernt **VAL** führende Leerzeichen, Tabulatoren und Zeilenvorschübe aus der Zeichenkette des Arguments. Zum Beispiel gibt

```
VAL("        -33/LP")
```

den Wert -33 an.

Vergleichen Sie auch

STR$

Beispiel

Das nachstehende Programm gibt die Namen und Adressen von Personen mit den jeweiligen Ortsnetzkennzahlen aus:

```
INPUT "Welches Ortsnetz suchen: "; Zielnetz
OPEN "ADRESS.DAT" FOR INPUT AS #1
DO WHILE NOT EOF(1)
   INPUT #1, Nm$, TeleNum$, Strasse$, Stadt$
   'VAL liest alles bis zum ersten nicht numerischen
   'Zeichen ("-" in diesem Fall).
   Netz = VAL(TeleNum$)
   IF Netz = Zielnetz THEN
      PRINT : PRINT Nm$
      PRINT Strasse$
      PRINT Stadt$
   END IF
LOOP
CLOSE : END
```

VARPTR-, VARSEG-Funktionen

Funktion

Gibt die Adresse einer Variablen an.

Syntax

VARPTR (*Variablenname*)

VARSEG (*Variablenname*)

Anmerkungen

Der *Variablenname* kann jede BASIC-Variable sein, einschließlich einer Verbundvariablen oder eines Verbundelementes.

Die Funktion **VARPTR** gibt eine vorzeichenlose Ganzzahl an, die der Offset der Variablen innerhalb ihres Segmentes ist. Die Funktion **VARSEG** gibt eine vorzeichenlose Ganzzahl an, die den Segmentteil der Variablen-Adresse darstellt. Wenn *Variablenname* nicht definiert ist, bevor **VARPTR** oder **VARSEG** aufgerufen werden, wird die Variable angelegt und ihre Adresse angegeben. Wenn *Variablenname* eine Zeichenkettenvariable ist, geben **VARPTR** und **VARSEG** den Speicherplatz des ersten Bytes des Zeichenkettenbeschreibers an.

Wichtig Da viele BASIC-Anweisungen den Speicherplatz von Variablen verändern, sollten Sie die von **VARPTR** und **VARSEG** angegebenen Werte direkt nach Verwendung der Funktionen benutzen. Siehe Abschnitt 2.3.3, " Speicherzuweisung für Variablen", für eine Aufzählung von Umständen, die eine Variablenbewegung verursachen.

VARPTR und **VARSEG** werden häufig mit **BLOAD, BSAVE, CALL ABSOLUTE, CALL INTERRUPT, PEEK, POKE** oder bei der Übergabe von Datenfeldern an Prozeduren, die in anderen Sprachen geschrieben sind, verwendet.

Um mit **VARPTR** oder **VARSEG** die Adresse eines Datenfeldes zu erhalten, verwenden Sie das erste Element des Datenfeldes als Argument:

```
DIM A(150)
.
.
.
FeldAdress=VARPTR(A(1))
```

Wichtig Um die Adresse eines Dateipuffers zu erhalten, dürfen Sie nicht mehr **VARPTR** verwenden. Benutzen Sie **FILEATTR**, um Informationen über eine Datei zu erhalten.

Darüber hinaus funktionieren in früheren BASIC-Versionen geschriebene Programme, die **VARPTR** verwenden, um auf große numerische Datenfelder zuzugreifen, nicht mehr. Sie müssen jetzt eine Kombination von **VARPTR** und **VARSEG** verwenden. Zum Beispiel arbeitet folgender Ausschnitt, geschrieben in QuickBASIC, Version 3.0, nicht mehr korrekt:

```
DIM Wuerfel(675)
.
.
.
BSAVE "graph.dat",VARPTR(Wuerfel(1)),2700
```

Der Ausschnitt würde neu geschrieben wie folgt aussehen:

```
DIM Wuerfel(675)
.
.
.
'Wechsele Segment in Segment, das Wuerfel enthält
DEF SEG=VARSEG(Wuerfel(1))
BSAVE "graph.dat",VARPTR(Wuerfel(1)),2700
'Stelle BASIC-Segment wieder her
DEF SEG
```

Sie können **VARPTR** auch allein benutzen, um die Adresse einer in **DGROUP** gespeicherten Variablen zu erhalten. Sie müssen beide, **VARPTR** und **VARSEG**, benutzen, um die komplette Adresse einer als weites Objekt "Far Object" gespeicherten Variablen zu bekommen. Siehe Abschnitt 2.3.3, "Speicherzuweisung für Variablen", für Regeln, die festlegen, wann eine Variable in **DGROUP** gespeichert wird.

Vergleichen Sie auch

BLOAD, BSAVE, CALL ABSOLUTE, CALL INTERRUPT, DEF SEG, PEEK, POKE

Beispiel

In den Beschreibungen zu **BLOAD, BSAVE, CALL ABSOLUTE** und **CALL INTERRUPT** finden Sie Beispiele, wie **VARPTR** und **VARSEG** mit diesen Anweisungen verwendet werden.

Das folgende Programm verdeutlicht, wie **VARPTR** und **VARSEG** verwendet werden, um ein BASIC-Datenfeld an eine C-Funktion zu übergeben. Die C-Funktion wird separat kompiliert und in eine Quick-Bibliothek eingefügt oder mit dem BASIC-Hauptprogramm gebunden. Weitere Informationen über die Programmierung in verschiedenen Sprachen finden Sie in Anhang C, "Aufruf von C- und Assembler-Routinen", in *Lernen und Anwenden von Microsoft QuickBASIC.*

```
' Ein BASIC-Programm, das ein Datenfeld an eine C-
' Funktion übergibt.
DECLARE SUB AddFeld CDECL (BYVAL Offs AS INTEGER, _
                           BYVAL Segm AS INTEGER, _
                           BYVAL Num AS INTEGER)
DIM A(1 TO 100) AS INTEGER
'Fülle das Datenfeld mit den Zahlen 1 bis 15.
FOR I=1 TO 15
   A(I)=I
NEXT I
' Aufruf der C-Funktion. AddFeld erwartet eine lange
' Adresse (Segment und Offset). Da CDECL Daten von
' rechts nach links auf dem Stapel ablegt, schreibe
' den Offset ( VARPTR (A(1)) ) zuerst in die Liste,
' gefolgt von dem Segment ( VARSEG (A(1)) ).
CALL AddFeld (VARPTR(A(1)),VARSEG(A(1)),15%)
'Gib das geänderte Datenfeld aus
FOR I=1 TO 15
   PRINT A(I)
NEXT I
END
```

```
/* Füge eins zu den ersten anz Elementen von Datenfeld
feld */
void far addfeld(feld,anz)
int far *feld;
int anz;
{
   int i;
   for(i=0;i<anz;i++) feld[i]++;
}
```

VARPTR$-Funktion

Funktion

Gibt die Adresse einer Variablen in Zeichenkettendarstellung zur Verwendung in den Anweisungen **DRAW** und **PLAY** an.

Syntax

VARPTR$ (*Variablenname*)

Anmerkungen

Der *Variablenname* ist der Name einer Variablen des Programms.

Wenn *Variablenname* ein Datenfeldelement ist, muß das Datenfeld dimensioniert sein, bevor die **VARPTR$**-Funktion verwendet wird. Das Datenfeld muß ein Datenfeld von Zeichenketten variabler Länge sein.

Hinweis Um korrekte Ergebnisse zu garantieren, verwenden Sie den Wert von **VARPTR$** direkt nach Aufruf dieser Funktion.

Unterschied zu BASICA

In QuickBASIC-Programmen muß **VARPTR$** bei den Anweisungen **DRAW** und **PLAY** zur Ausführung von Teilzeichenketten, die Variablen enthalten, verwendet werden.

BASICA unterstützt sowohl die Syntax von **VARPTR$** als auch die Syntax, die lediglich den Variablennamen enthält.

Vergleichen Sie auch

DRAW, PLAY

Beispiel

Vergleichen Sie die Beispiele auf den Nachschlageseiten für **DRAW** und **PLAY**.

VIEW-Anweisung

Funktion

Definiert die Bildschirmgrenzen für die grafische Ausgabe.

Syntax

VIEW [[**SCREEN**](*x1,y1*)-(*x2,y2*)[,[*Farbe*][,*Rand*]]]

Anmerkungen

Die nachstehende Liste beschreibt die Teile der **VIEW**-Anweisung:

Teil	*Beschreibung*
SCREEN	Wenn **SCREEN** verwendet wird, sind die x- und y-Koordinaten absolut in Bezug auf den Bildschirm, nicht relativ zum Rand des physikalischen Darstellungsfeldes. Nur Graphen innerhalb des Darstellungsfeldes werden gezeichnet. Wenn **SCREEN** ausgelassen wird, werden alle Punkte relativ zu dem Darstellungsfeld (vor dem Zeichnen des Punktes werden *x1* und *x2* Koordinaten hinzuaddiert) gezeichnet.
(*x1,y1*)-(*x2,y2*)	Legt einen rechtwinkligen Bereich auf dem Bildschirm fest. Die Platzhalter *x1*, *y1*, *x2*, und *y2* sind numerische Ausdrücke, die die Koordinaten der sich diagonal gegenüberliegenden Ecken des Bereiches darstellen.
Farbe	Das Farbattribut der Farbe, mit der der Bereich ausgemalt wird. Beim Auslassen von *Farbe* wird der Bereich nicht ausgemalt.
Rand	Jeder numerische Ausdruck in diesem Bereich zeichnet eine Linie um das Darstellungsfeld, wenn Platz vorhanden ist. Wenn *Rand* ausgelassen wird, wird kein Rand gezeichnet.

Die **VIEW**-Anweisung definiert ein "physikalisches Darstellungsfeld" oder einen rechteckigen Ausschnitt auf dem Bildschirm, in dem die Grafiken abgebildet werden können. Alle in der Anweisung verwendeten Koordinaten müssen innerhalb der physikalischen Grenzen des Bildschirms liegen.

Die Angabe von **VIEW** ohne Argument definiert den gesamten Bildschirm als Darstellungsfeld. **RUN** und **SCREEN** definieren ebenfalls den gesamten Bildschirm als Darstellungsfeld und deaktivieren somit alle mit **VIEW** definierten Darstellungsfelder.

Beispiele

Das folgende Beispiel stellt die Ausgaben nach der Ausführung von VIEW und VIEW SCREEN gegenüber:

```
SCREEN 2
'Die Option SCREEN wandelt alle Bildschirmkoordinaten
'in absolute Koordinaten um.
VIEW SCREEN(10,10)-(200,100)
'Diese Gerade ist nicht sichtbar, weil ihre Endpunkte
'außerhalb des Darstellungsfeldes liegen.
LINE (5,5)-(100,5)
PRINT "Weiter mit jeder Taste."
SUSPEND$ = INPUT$(1) 'Warte auf eine Tasteneingabe.
'Keine SCREEN-Option - alle Bildschirmkoordinaten relativ zu
'diesem Darstellungsfeld:
VIEW (10,10)-(200,100)
'Diese Gerade ist jetzt sichtbar, weil ihre Endpunkte
'zu (10,10) addiert werden.
LINE (5,5)-(100,5)
```

Sie können mehrere **VIEW**-Anweisungen benutzen. Wenn das neu beschriebene Darstellungsfeld nicht ganz innerhalb des vorhergehenden Darstellungsfeldes liegt, kann der Bildschirm mit der Anweisung **VIEW** neu initialisiert werden. Anschließend kann das neue Darstellungsfeld festgelegt werden. Liegt das neue Darstellungsfeld vollständig innerhalb des vorhergehenden Feldes, wie im folgenden Beispiel, so ist die mittlere **VIEW**-Anweisung nicht erforderlich. Dieses Beispiel eröffnet drei Darstellungsfelder, jedes kleiner als das vorhergehende. In jedem Fall werden die Punkte der Geraden, die außerhalb der Begrenzung des Darstellungsfeldes liegen, abgeschnitten und erscheinen nicht auf dem Bildschirm.

```
SCREEN 1
CLS
VIEW    'Mache den überwiegenden Teil des Bildschirms
        'zum Darstellungsfeld
VIEW (10,10)-(300,180),,1
   CLS
   LINE (0,0) - (310,190),1
   LOCATE 1,11: PRINT "Ein großes Darstellungsfeld"
```

```
VIEW SCREEN (50,50)-(250,150),,1
   CLS  'Beachte, daß CLS nur das Darstellungsfeld
        'löscht
   LINE (300,0)-(0,199),1
   LOCATE 9,9
   PRINT "Ein mittelgroßes Darstellungsfeld"
VIEW SCREEN (80,80)-(200,125),,1
   CLS
   CIRCLE (150,100),20,1
   LOCATE 11,9: PRINT "Ein kleines Darstellungsfeld"
```

VIEW PRINT-Anweisung

Funktion

Setzt die Grenzen des Bildschirm-Textfensters.

Syntax

VIEW PRINT [*Kopfzeile* **TO** *Fußzeile*]

Anmerkungen

Das Argument *Kopfzeile* ist die Nummer der obersten Zeile im Fenster, *Fußzeile* ist die Nummer der untersten Zeile.

Ohne die Parameter *Kopfzeile* und *Fußzeile* initialisiert die **VIEW PRINT**-Anweisung den gesamten Bildschirmbereich als Textfenster. Die Anzahl der Zeilen des Bildschirms hängt von dem Bildschirmmodus und davon ab, ob die Option **/h** beim Start von QuickBASIC verwendet wurde oder nicht. Weitere Informationen finden Sie unter der Beschreibung der Anweisung **WIDTH** und im Kapitel 3, "Datei- und Geräte-E/A", in *Programmieren in BASIC: Ausgewählte Themen.*

Anweisungen und Funktionen, die innerhalb des definierten Textfensters wirksam sind, umfassen **CLS, LOCATE, PRINT** und die **SCREEN**-Funktion.

Vergleichen Sie auch

CLS, LOCATE, PRINT, SCREEN-Funktion, VIEW

Beispiel

Siehe Beispiel für **CLS**.

WAIT-Anweisung

Funktion

Unterbricht die Programmausführung, während der Status eines Maschinen-Eingabeanschlusses (Input Port) überwacht wird.

Syntax

WAIT *Anschlußnummer*, *AND-Ausdruck*[, *XOR-Ausdruck*]

Anmerkungen

Die folgende Tabelle beschreibt die Argumente der **WAIT**-Anweisung:

Argument	*Beschreibung*
Anschlußnummer	Ein ganzzahliger Ausdruck von 0 bis 255, der die Nummer des Anschlusses darstellt.
AND-Ausdruck	Ein ganzzahliger Ausdruck, der mit den Daten von dem Anschluß durch eine **AND**-Operation verknüpft wird.
XOR-Ausdruck	Ein ganzzahliger Ausdruck, der mit den Daten von dem Anschluß durch eine **XOR**-Operation verknüpft wird.

Die Anweisung **WAIT** unterbindet die Ausführung, bis ein bestimmtes Bit-Muster von einem festlegten Eingabeanschluß gelesen wird. Das von dem Anschluß gelesene Datum wird mit *XOR-Ausdruck*, sofern vorhanden, unter Verwendung einer **XOR**-Operation verbunden. Das Ergebins wird anschließend unter Verwendung einer **AND**-Operation mit dem *AND-Ausdruck* verknüpft. Wenn das Ergebnis Null ist, verzweigt BASIC zurück und liest das Datum am Anschluß erneut. Ist das Ergebnis ungleich Null, so wird die Ausführung mit der nächsten Anweisung fortgesetzt. Wird *XOR-Ausdruck* nicht angegeben, so wird dafür 0 angenommen.

Warnung Es ist möglich, mit der Anweisung **WAIT** eine Endlosschleife zu erzeugen, wenn der Eingabeanschluß kein Bit-Muster ungleich Null erzeugen kann. In diesem Fall müssen Sie die Maschine erneut manuell starten.

Beispiel

```
WAIT 32,2
```

WHILE...WEND-Anweisung

Funktion

Führt eine Folge von Anweisungen in einer Schleife aus, solange eine angegebene Bedingung wahr ist.

Syntax

WHILE *Bedingung*
.
.
.
[*Anweisungen*]
.
.
.
WEND

Anmerkungen

Wenn die *Bedingung* wahr ist (d. h. wenn sie ungleich Null ist), werden alle folgenden Anweisungen solange ausgeführt, bis die Anweisung **WEND** erreicht wird. BASIC kehrt dann zu der **WHILE**-Anweisung zurück und überprüft *Bedingung*. Wenn *Bedingung* weiterhin wahr ist, wird die Schleife abermals ausgeführt. Wenn *Bedingung* nicht mehr wahr ist (oder wenn sie gleich Null ist), wird die Ausführung mit der auf die **WEND**-Anweisung folgenden Anweisung fortgesetzt.

Hinweis Die QuickBASIC-Anweisung **DO...LOOP** bietet eine umfassendere und flexiblere Struktur zur Schleifensteuerung.

WHILE...WEND-Schleifen können beliebig verschachtelt werden. Jedes **WEND** gehört zum jeweils letzten **WHILE**. Eine nicht abgeschlossene **WHILE**-Anweisung führt zu der `WHILE ohne WEND`-Fehlermeldung. Eine **WEND**-Anweisung ohne dazugehöriges **WHILE** führt zu der `WEND ohne WHILE`-Fehlermeldung.

Hinweis Verzweigen Sie nicht in das Innere einer **WHILE...WEND**-Schleife, ohne das **WHILE** auszuführen. Dies kann Laufzeitfehler hervorrufen oder Programmprobleme erzeugen, die schwierig zu finden sind.

Vergleichen Sie auch

DO...LOOP

Beispiel

Der folgende Ausschnitt führt einen Bubblesort mit dem Datenfeld `A$` durch. In der vierten Zeile wird die Variable `TAUSCHE` durch die Zuweisung eines Nicht-Null-Wertes wahr; dies führt zu mindestens einem Durchlauf der **WHILE...WEND**-Schleife (in einer **DO...LOOP**-Anweisung ist diese Konstruktion unnötig). Wenn kein weiterer Tausch mehr erfolgt, sind alle Elemente von `A$` sortiert. `Tausche` ist falsch (gleich Null) und das Programm fährt mit der auf `WEND` folgenden Zeile fort.

```
'Führe Bubblesort mit Datenfeld A$ durch.
CONST FALSE=0, TRUE=NOT FALSE
Max = UBOUND(A$)
Tausche=TRUE  'Führe ersten Schleifendurchlauf durch.
WHILE Tausche  'Sortiere, bis keine Elemente mehr
               'getauscht werden.
     Tausche=FALSE
     ' Vergleiche die Datenfeldelemente paarweise.
     ' Wenn zwei getauscht sind, wird der nächste
     ' Schritt erzwungen, indem Tausche TRUE gesetzt
     ' wird.
     FOR I = 2 TO Max
        IF A$(I-1) > A$(I) THEN
           Tausche=TRUE
           SWAP A$(I-1), A$(I)
        END IF
     NEXT
WEND
.
.
.
```

WIDTH-Anweisung

Funktion

Weist die Breite einer Ausgabezeile einer Datei oder einem Gerät zu, oder ändert die Anzahl von Spalten und Zeilen, die auf dem Bildschirm angezeigt werden.

Syntax

WIDTH [*Spalten*][,*Zeilen*]

WIDTH {# *Dateinummer* | *Gerät*}, *Breite*

WIDTH LPRINT *Breite*

Anmerkungen

Sowohl Dateien als auch Geräten kann die Breite einer Ausgabezeile zugewiesen werden. Die folgende Liste beschreibt die verschiedenen Formen der **WIDTH**-Anweisung:

Syntax	***Beschreibung***
WIDTH [*Spalten*][, *Zeilen*]	Setzt die Anzahl der auf dem Bildschirm anzuzeigenden Spalten und Zeilen. Der Wert von *Spalten* muß entweder 40 oder 80 sein. Der Standardwert ist 80. Der Wert von *Zeilen* kann, abhängig vom verwendeten Bildschirmadapter und Bildschirmmodus, 25, 30, 43, 50 oder 60 sein. Unter der Beschreibung der **SCREEN**-Anweisung finden Sie weitere Informationen. Der Standardwert der Anzahl der Zeilen ergibt sich aus dem Wert beim Programmstart.
WIDTH *#Dateinummer, Breite*	Setzt die Zeilenbreite eines als Datei geöffneten Ausgabegerätes (zum Beispiel, **LPT1:** oder **CONS:**) auf *Breite*. Die *Dateinummer* ist die Nummer, die mit der Datei in der **OPEN**-Anweisung verbunden wurde. Diese Form erlaubt eine Veränderung der Breite, während eine Datei geöffnet ist, da diese Anweisung sofort ausgeführt wird.

Syntax	*Beschreibung*
WIDTH *Gerät,Breite*	Setzt die Zeilenbreite von *Gerät* (ein Geräte-Dateiname) auf *Breite*. Das *Gerät* sollte ein Zeichenkettenausdruck sein (z.B. "CONS:"). Beachten Sie, daß die Breitenzuweisung bis zur nächsten, das Gerät betreffende, **OPEN**-Anweisung *verzögert* wird; die Zuweisung betrifft nicht die Ausgaben für eine bereits geöffnete Datei.
WIDTH LPRINT *Breite*	Setzt die Zeilenbreite des Zeilendruckers auf *Breite*. Nachfolgende **LPRINT**-Anweisungen benutzen diese Breite.

Vergleichen Sie auch

SCREEN-Anweisung

Beispiel

Im folgenden Beispiel wird die Satzbreite für Datei #1 (der Zeilendrucker) auf verschiedene Breiten gesetzt:

```
OPEN "LPT1:" FOR OUTPUT AS #1
Test$ = "1234567890"
WIDTH #1, 3
PRINT #1, Test$
WIDTH #1, 4
PRINT #1, Test$
CLOSE
```

Ausgabe

```
123
456
789
0
1234
5678
90
```

WINDOW-Anweisung

Funktion

Definiert die logischen Dimensionen des aktuellen Darstellungsfeldes.

Syntax

WINDOW [[SCREEN](*x1*,*y1*)-(*x2*,*y2*)]

Anmerkungen

Die Anweisung **WINDOW** erlaubt es dem Benutzer, eigene Koordinatensysteme zum Zeichnen von Geraden, Graphen oder Objekten zu erstellen, ohne durch die physikalischen Koordinaten des Bildschirms (die Dimensionen des Bildschirms) behindert zu werden. Dies wird durch Neudefinition der Bildschirm-Randkoordinaten mit den "logischen Koordinaten" (*x1*,*y1*) und (*x2*,*y2*) erreicht. Diese logischen Koordinaten sind Zahlen einfacher Genauigkeit.

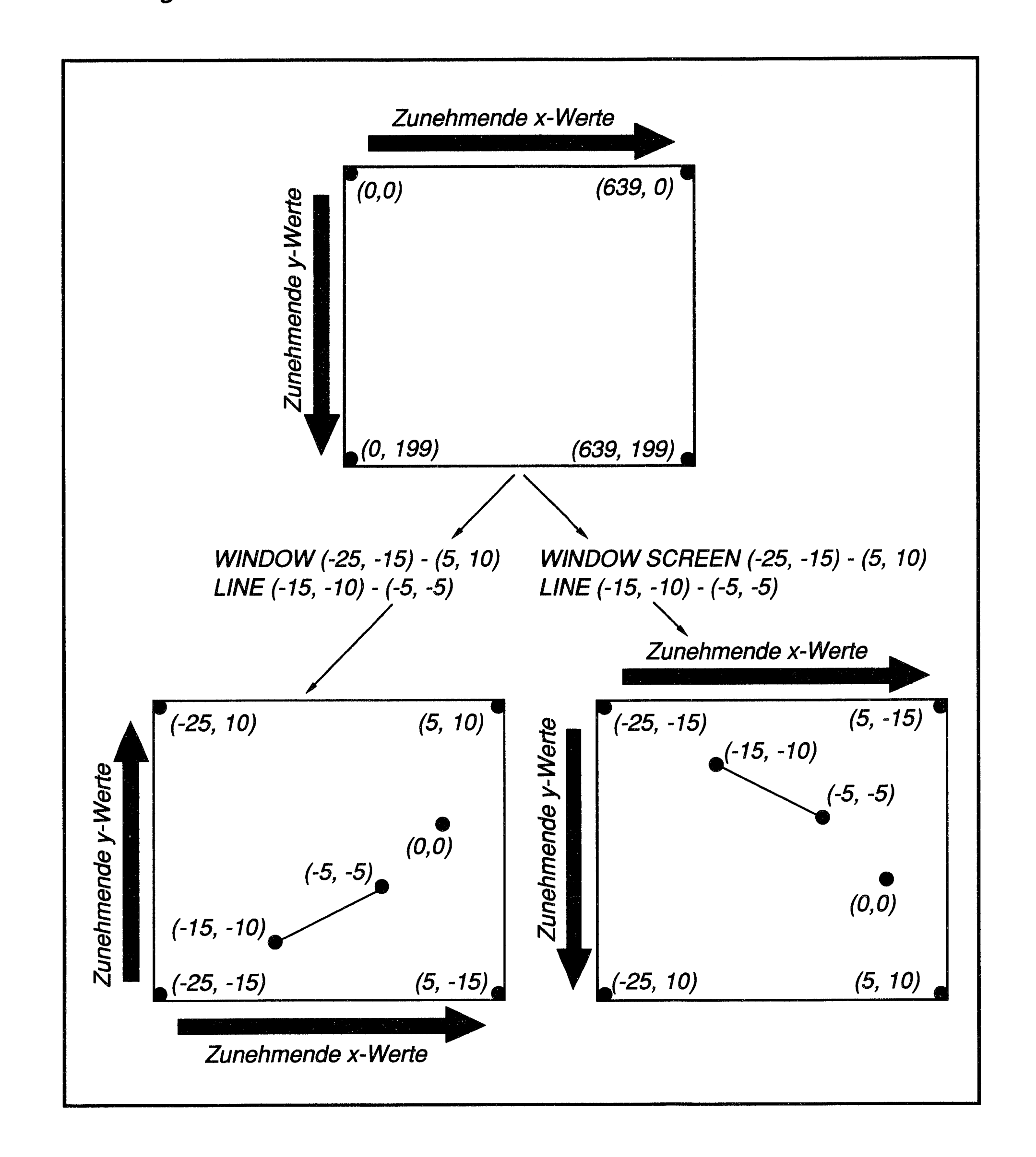
WINDOW definiert den Abschnitt des logischen Koordinatensystems, der auf den physikalischen Koordinaten des Bildschirms abgebildet wird. Alle folgenden Grafik-Anweisungen verwenden diese neuen logischen Koordinaten und werden innerhalb des aktuellen Darstellungsfeldes angezeigt. (Die Größe des Darstellungsfeldes kann mit der Anweisung **VIEW** verändert werden.)

RUN oder **WINDOW** ohne Argumente schaltet die Fenster-Transformation aus.

Die Variante **WINDOW SCREEN** kehrt die normale kartesische Richtung der y-Koordinate um, so daß y-Werte vom Negativen ins Positive von oben nach unten gehen.

Abbildung R.14 zeigt die Auswirkungen von **WINDOW** und **WINDOW SCREEN** auf eine im Bildschirmmodus 2 gezeichnete Gerade. Beachten Sie die Veränderung der Koordinaten in den Bildschirmecken.

Abbildung R.14 WINDOW und WINDOW SCREEN



Beispiele

Das folgende Beispiel zeigt zwei Geraden mit den gleichen Endpunkt-Koordinaten. Die erste wird im Standard-Bildschirm gezeichnet, die zweite in einem neu definierten Fenster.

```
SCREEN 2
LINE (100,100) - (150,150), 1
LOCATE 2,20
PRINT "Die Gerade auf dem Standard-Bildschirm"
WINDOW SCREEN (100,100) - (200,200)
LINE (100,100) - (150,150), 1
LOCATE 8,18
PRINT"& die selbe Gerade auf dem neu definierten";
PRINT" Fenster"
```

Das folgende Beispiel zeigt, wie durch Veränderung der Fenstergröße die Größe einer auf dem Bildschirm gezeichneten Figur verändert wird. Dies ergibt einen Zoom-Effekt: Wenn das Fenster kleiner wird, erscheint die Figur auf dem Bildschirm größer, bis schließlich Teile von ihr abgeschnitten werden, da sie außerhalb des Fensters liegen. Wenn das Fenster größer wird, erscheint die Figur kleiner auf dem Bildschirm.

```
PRINT "Betätigen Sie EINGABETASTE zum Starten.";
INPUT;"",A$
SCREEN 1 : COLOR 7          'Grauer Bildschirm
X = 500 : Xdelta = 50
DO
   DO WHILE X < 525 AND X > 50
      X = X + Xdelta       'Verändere die Fenstergröße
      CALL Zoom(X)
      FOR I = 1 TO 1000    'Verzögerungsschleife
         IF INKEY$ <> "" THEN END 'Halte, wenn Taste
                                  'betätigt wird
      NEXT
   LOOP
   X = X - Xdelta
   Xdelta = -Xdelta        'Drehe Größenveränderung um
LOOP
```

```
SUB Zoom(X) STATIC
   CLS
   WINDOW (-X, -X) - (X,X)   'Definiere neues Fenster
   LINE (-X,-X) - (X,X) ,1,B  'Zeichne Fensterrand
   CIRCLE (0,0),60,1,,,.5     'Zeichne Ellipse mit x-
                              'Radius 60
   PAINT (0,0),1              'Male Ellipse aus
END SUB
```

WRITE-Anweisung

Funktion

Gibt Daten auf dem Bildschirm aus.

Syntax

WRITE [*Ausdrucksliste*]

Anmerkungen

Wenn *Ausdrucksliste* ausgelassen ist, wird eine Leerzeile geschrieben. Wenn *Ausdrucksliste* angegeben ist, werden die Werte der Ausdrücke auf den Bildschirm geschrieben. Die Ausdrücke der Liste können numerische und/oder Zeichenkettenausdrücke sein. Sie müssen durch Kommata getrennt werden.

Wenn die Werte der Ausdrücke geschrieben werden, wird jeder Ausdruck von dem vorhergehenden durch ein Komma getrennt. Geschriebene Zeichenketten werden durch Anführungszeichen begrenzt. Nachdem der letzte Wert in der Liste ausgegeben wurde, fügt BASIC die Folge Wagenrücklauf + Zeilenvorschub an.

Die **WRITE**-Anweisung schreibt numerische Werte ohne führende oder nachfolgende Leerzeichen.

Vergleichen Sie auch

PRINT

Beispiel

Das folgende Beispiel zeigt den Unterschied zwischen den Anweisungen **PRINT** und **WRITE**:

```
A=80 : B=90 : C$="Das war's!" : D=-1.0E-13
WRITE A,B,C$,D
PRINT A,B,C$,D
```

Ausgabe

```
80,90,"Das war's!",-1E-13
  80          90          Das war's!  -1E-13
```

WRITE #-Anweisung

Funktion

Schreibt Daten in eine sequentielle Datei.

Syntax

WRITE # *Dateinummer, Ausdrucksliste*

Anmerkungen

Die *Dateinummer* ist die Nummer, die in der **OPEN**-Anweisung benutzt wurde. Die Datei muß im **OUTPUT**- oder **APPEND**-Modus geöffnet worden sein. Die Ausdrücke in der *Ausdrucksliste* sind Zeichenkettenausdrücke und/oder numerische Ausdrücke, die durch Kommata getrennt werden. Falls Sie *Ausdrucksliste* auslassen, schreibt die **WRITE #**-Anweisung eine leere Zeile in die Datei.

Im Unterschied zu der **PRINT #**-Anweisung fügt die **WRITE #**-Anweisung Kommata zwischen den Daten ein, während sie in die Datei geschrieben werden. Es ist nicht nötig, explizite Begrenzer in die Liste zu schreiben. Nachdem das letzte Datum der Liste in die Datei geschrieben wurde, wird eine neue Zeile eingefügt.

Wenn mit einer **WRITE #**-Anweisung versucht wird, Daten in eine sequentielle Datei zu schreiben, die durch eine **LOCK**-Anweisung gesperrt ist, erscheint die Fehlermeldung `Zugriff nicht gestattet`, es sei denn, der Fehler wurde bereits durch das Programm verfolgt. Jede der normalen BASIC-Fehlerbehandlungsroutinen kann diesen Fehler verfolgen und untersuchen.

Vergleichen Sie auch

LOCK, OPEN, PRINT #, WRITE

Beispiel

Die Ausgabe des folgenden Programms stellt die Unterschiede zwischen den Anweisungen **WRITE #** und **PRINT #** dar:

```
A$ = "AEG, elektrischer Gartenzwerg" : B$ = "299.00 DM"
OPEN "PREISE" FOR OUTPUT AS #1   'Öffne PREISE zum
                                 'Schreiben
PRINT #1,A$,B$     'Speichere A$ und B$ in erstem Satz
                   'mit PRINT #
WRITE #1,A$,B$     'Speichere A$ und B$ in zweitem Satz
                   'mit WRITE #
CLOSE #1
```

Ausgabe

```
AEG                     elektrischer Gartenzwerg 299.00 DM
"AEG, elektrischer Gartenzwerg","299.00 DM"
```

Teil 3: Anhänge



Anhang A ASCII-Zeichencodes und Tastaturabfragecodes

A.2 BASIC-Befehlsverzeichnis

Die folgende Tabelle gibt Ihnen einen Überblick über die in diesem Handbuch verwendeten Tastennamen und den Entsprechungen, die Sie auf einigen Tastaturen an deren Stelle finden.

ALT-TASTE	ALT
RÜCKTASTE	BACKSPACE, BKSP
UNTBR-TASTE	BREAK
UMSCHALT-FESTSTELLTASTE	CAPS LOCK
STRG-TASTE	CONTROL, CTRL
RICHTUNGSTASTEN	CURSOR KEYS
ENTF-TASTE	DELETE, DEL
NACH UNTEN (↓)	DOWN
ENDE-TASTE	END
EINGABETASTE	ENTER
ESC-TASTE	ESCAPE, ESC
POS1-TASTE	HOME
EINFG-TASTE	INSERT, INS
NACH LINKS (←)	LEFT
NUM-FESTSTELLTASTE	NUMLOCK
BILD ↓-TASTE	PAGE DOWN, PG DN
BILD ↑-TASTE	PAGE UP, PG UP
PAUSE-TASTE	PAUSE
ROLLEN-FESTSTELLTASTE	SCROLL LOCK
UMSCHALTTASTE	SHIFT
LEERTASTE	SPACE BAR
S-ABF-TASTE	SYS RQ
DRUCK-TASTE	PRINT SCREEN, PRTSC
NACH RECHTS (→)	RIGHT
TAB-TASTE	TAB
NACH OBEN (↑)	UP

A.1 Tastaturabfragecodes

Die folgende Tabelle zeigt die DOS Tastaturabfragecodes. Diese Codes werden durch die Funktion **INKEY$** zurückgegeben.

Tastaturkombinationen mit NUL in der Zeichen-Spalte ergeben zwei Bytes — ein Null-Byte (&H00) gefolgt von dem Wert in der Dez- und Hex-Spalte. Zum Beispiel erhalten Sie, wenn Sie ALT+F1 drücken, ein Null-Byte gefolgt von dem Byte mit dem Wert 104 (&H68).

Hinweis Benutzen Sie diese Codes nicht zur Verfolgung. Die Einträge in diesem Handbuch für die **KEY**(*n*)-Anweisungen beschreiben die korrekten Codes für die Verfolgung von gedrückten Tasten.

Taste	Abfrage-code		ASCII oder Erweitert*			ASCII oder Erweitert* mit UMSCHALT			ASCII oder Erweitert* mit STRG			ASCII oder Erweitert* mit ALT		
	Dez	Hex	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei
ESC	1	01	27	1B		27	1B		27	1B				
1 !	2	02	49	31	1	33	21	!				120	78	NUL
2 @	3	03	50	32	2	64	40	@	3	03	NUL	121	79	NUL
3 #	4	04	51	33	3	35	23	#				122	7A	NUL
4 $	5	05	52	34	4	36	24	$				123	7B	NUL
5 %	6	06	53	35	5	37	25	%				124	7C	NUL
6 ^	7	07	54	36	6	94	5E	^	30	1E		125	7D	NUL
7 &	8	08	55	37	7	38	26	&				126	7E	NUL
8 *	9	09	56	38	8	42	2A	*				127	7F	NUL
9 (	10	0A	57	39	9	40	28	(				128	80	NUL
0)	11	0B	48	30	0	41	29	)				129	81	NUL
- _	12	0C	45	2D	-	95	5F	_	31	1F		130	82	NUL
= +	13	0D	61	3D	=	43	2B	+				131	83	NUL
BKSP	14	0E	8	08		8	08		127	7F				
TAB	15	0F	9	09		15	0F	NUL						
Q	16	10	113	71	q	81	51	Q	17	11		16	10	NUL
W	17	11	119	77	w	87	57	W	23	17		17	11	NUL
E	18	12	101	65	e	69	45	E	5	05		18	12	NUL
R	19	13	114	72	r	82	52	R	18	12		19	13	NUL
T	20	14	116	74	t	84	54	T	20	14		20	14	NUL
Y	21	15	121	79	y	89	59	Y	25	19		21	15	NUL
U	22	16	117	75	u	85	55	U	21	15		22	16	NUL
I	23	17	105	69	i	73	49	I	9	09		23	17	NUL
O	24	18	111	6F	o	79	4F	O	15	0F		24	18	NUL
P	25	19	112	70	p	80	50	P	16	10		25	19	NUL
[{	26	1A	91	5B	[	123	7B	{	27	1B				
] }	27	1B	93	5D	]	125	7D	}	29	1D				
ENTER	28	1C	13	0D	CR	13	0D	CR	10	0A	LF			
CTRL	29	1D												
A	30	1E	97	61	a	65	41	A	1	01		30	1E	NUL
S	31	1F	115	73	s	83	53	S	19	13		31	1F	NUL
D	32	20	100	64	d	68	44	D	4	04		32	20	NUL
F	33	21	102	66	f	70	46	F	6	06		33	21	NUL
G	34	22	103	67	g	71	47	G	7	07		34	22	NUL
H	35	23	104	68	h	72	48	H	8	08		35	23	NUL
J	36	24	106	6A	j	74	4A	J	10	0A		36	24	NUL
K	37	25	107	6B	k	75	4B	K	11	0B		37	25	NUL
L	38	26	108	6C	l	76	4C	L	12	0C		38	26	NUL
; :	39	27	59	3B	;	58	3A	:						
' "	40	28	39	27	'	34	22	"						
` ~	41	29	96	60	`	126	7E	~						

* Erweiterter Code gibt NUL (ASCII 0) als erstes Zeichen zurück. Dieses ist ein Signal, daß ein zweiter (erweiterter) Code im Tastaturpuffer wartet.

Taste	Abfrage-code		ASCII oder Erweitert*			ASCII oder Erweitert* mit UMSCHALT			ASCII oder Erweitert* mit STRG			ASCII oder Erweitert* mit ALT		
	Dez	Hex	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei	Dez	Hex	Zei
L SHIFT	42	2A												
\ |	43	2B	92	5C	\	124	7C	|	28	1C				
Z	44	2C	122	7A	z	90	5A	Z	26	1A		44	2C	NUL
X	45	2D	120	78	x	88	58	X	24	18		45	2D	NUL
C	46	2E	99	63	c	67	43	C	3	03		46	2E	NUL
V	47	2F	118	76	v	86	56	V	22	16		47	2F	NUL
B	48	30	98	62	b	66	42	B	2	02		48	30	NUL
N	49	31	110	6E	n	78	4E	N	14	0E		49	31	NUL
M	50	32	109	6D	m	77	4D	M	13	0D		50	32	NUL
, <	51	33	44	2C	,	60	3C	<						
. >	52	34	46	2E	.	62	3E	>						
/ ?	53	35	47	2F	/	63	3F	?						
R SHIFT	54	36												
* PRTSC	55	37	42	2A	*		INT 5**		16	10				
ALT	56	38												
SPACE	57	39	32	20	SPC	32	20	SPC	32	20	SPC	32	20	SPC
CAPS	58	3A												
F1	59	3B	59	3B	NUL	84	54	NUL	94	5E	NUL	104	68	NUL
F2	60	3C	60	3C	NUL	85	55	NUL	95	5F	NUL	105	69	NUL
F3	61	3D	61	3D	NUL	86	56	NUL	96	60	NUL	106	6A	NUL
F4	62	3E	62	3E	NUL	87	57	NUL	97	61	NUL	107	6B	NUL
F5	63	3F	63	3F	NUL	88	58	NUL	98	62	NUL	108	6C	NUL
F6	64	40	64	40	NUL	89	59	NUL	99	63	NUL	109	6D	NUL
F7	65	41	65	41	NUL	90	5A	NUL	100	64	NUL	110	6E	NUL
F8	66	42	66	46	NUL	91	5B	NUL	101	65	NUL	111	6F	NUL
F9	67	43	67	43	NUL	92	5C	NUL	102	66	NUL	112	70	NUL
F10	68	44	68	44	NUL	93	5D	NUL	103	67	NUL	113	71	NUL
NUM	69	45												
SCROLL	70	46												
HOME	71	47	71	47	NUL	55	37	7	119	77	NUL			
UP	72	48	72	48	NUL	56	38	8						
PGUP	73	49	73	49	NUL	57	39	9	132	84	NUL			
GREY -	74	4A	45	2D	-	45	2D	-						
LEFT	75	4B	75	4B	NUL	52	34	4	115	73	NUL			
CENTER	76	4C				53	35	5						
RIGHT	77	4D	77	4D	NUL	54	36	6	116	74	NUL			
GREY +	78	4E	43	2B	+	43	2B	+						
END	79	4F	79	4F	NUL	49	31	1	117	75	NUL			
DOWN	80	50	80	50	NUL	50	32	2						
PGDN	81	51	81	51	NUL	51	33	3	118	76	NUL			
INS	82	52	82	52	NUL	48	30	0						
DEL	83	53	83	53	NUL	46	2E	.						

** Unter DOS verursacht SHIFT+PRTSCR einen Interrupt-5, welcher den Bildschirminhalt druckt, bis ein Interrupt-Handler zur Ersetzung des Interrupt-5-Handlers definiert wurde.

A.2 ASCII-Zeichencodes

STRG	Dez	Hex	Zei	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dez	Hex	Zei
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	32	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dez	Hex	Zei
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dez	Hex	Zei
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	¦
125	7D	}
126	7E	~
127	7F	Δ *

* Der ASCII-Code 127 hat den Code DEL. Unter DOS hat dieser Code denselben Effekt wie ASCII 8 (BS). Der DEL-Code kann durch CTRL+BKSP erzeugt werden.

Dez	Hex	Zei
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dez	Hex	Zei
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dez	Hex	Zei
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dez	Hex	Zei
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
232	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	



Anhang B Reservierte Wörter in QuickBASIC

Die folgende Liste enthält die von Microsoft BASIC reservierten Wörter:

ABS	ACCESS	ACS	ALIAS
AND	ANY	APPEND	AS
ATN	BASE	BEEP	BINARY
BLOAD	BSAVE	BYVAL	CALL
CALLS	CASE	CDBL	CDECL
CHAIN	CHDIR	CHR$	CINT
CIRCLE	CLEAR	CLNG	CLOSE
CLS	COLOR	COM	COMMAND$
COMMON	CONST	COS	CSNG
CSRLIN	CVD	CVDMBF	CVI
CVL	CVS	CVSMBF	DATA
DATE$	DECLARE	DEF	DEFDBL
DEFINT	DEFLNG	DEFSNG	DEFSTR
DIM	DO	DOUBLE	DRAW
ELSE	ELSEIF	END	ENDIF
ENVIRON	ENVIRON$	EOF	EQV
ERASE	ERDEV	ERDEV$	ERL
ERR	ERROR	EXIT	EXP
FIELD	FILEATTR	FILES	FIX
FOR	FRE	FREEFILE	FUNCTION
GET	GOSUB	GOTO	HEX$
IF	IMP	INKEY$	INP

INPUT	INPUT$	INSTR	INT
INTEGER	IOCTL	IOCTL$	IS
KEY	KILL	LBOUND	LCASE$
LEFT$	LEN	LET	LINE
LIST	LOC	LOCAL	LOCATE
LOCK	LOF	LOG	LONG
LOOP	LPOS	LPRINT	LSET
LTRIM$	MID$	MKD$	MKDIR
MKDMBF$	MKI$	MKL$	MKS$
MKSMBF$	MOD	NAME	NEXT
NOT	OCT$	OFF	ON
OPEN	OPTION	OR	OUT
OUTPUT	PAINT	PALETTE	PCOPY
PEEK	PEN	PLAY	PMAP
POINT	POKE	POS	PRESET
PRINT	PSET	PUT	RANDOM
RANDOMIZE	READ	REDIM	REM
RESET	RESTORE	RESUME	RETURN
RIGHT$	RMDIR	RND	RSET
RTRIM$	RUN	SADD	SCREEN
SEEK	SEG	SELECT	SETMEM
SGN	SHARED	SHELL	SIGNAL
SIN	SINGLE	SLEEP	SOUND
SPACE$	SPC	SQR	STATIC
STEP	STICK	STOP	STR$
STRIG	STRING	STRING$	SUB
SWAP	SYSTEM	TAB	TAN
THEN	TIME$	TIMER	TO
TROFF	TRON	TYPE	UBOUND
UCASE$	UNLOCK	UNTIL	USING
VAL	VARPTR	VARPTR$	VARSEG
VIEW	WAIT	WEND	WHILE
WIDTH	WINDOW	WRITE	XOR

Anhang C Metabefehle

Dieser Anhang beschreibt die QuickBASIC-Metabefehle - Befehle, die QuickBASIC anweisen, Ihr Programm in einer ganz bestimmten Art und Weise zu behandeln.

Der erste Abschnitt beschreibt das Format, das für Metabefehle verwendet wird. Die nächsten zwei Abschnitte beschreiben bestimmte Metabefehle.

Durch Verwenden von Metabefehlen können Sie die folgenden Optionen ausführen:

- andere BASIC-Quelldateien an bestimmten Stellen während der Kompilierung einlesen und kompilieren (**$INCLUDE**).
- die Zuordnung von dimensionierten Datenfeldern steuern (**$STATIC** und **$DYNAMIC**).

C.1 Die Metabefehlssyntax

Metabefehle beginnen mit einem Dollarzeichen ($) und sind immer in einen Programmkommentar eingeschlossen. In einem Kommentar kann mehr als ein Metabefehl stehen. Mehrere Metabefehle werden durch die Leerraumzeichen - Leerzeichen, Tabulatoren oder Zeilenvorschub - voneinander getrennt. Bei Metabefehlen, denen Argumente folgen, steht zwischen dem Befehl und dem Argument ein Doppelpunkt:

REM *$METABEFEHL: Argument*

Zeichenkettenargumente müssen in einfache Anführungszeichen gesetzt werden. Leerraumzeichen zwischen den Elementen eines Metabefehls werden übergangen. Dies sind gültige Formate für Metabefehle:

```
REM $METABEFEHL1 $METABEFEHL2
REM $METABEFEHL1 : 'ZeiKett-Argument' $METABEFEHL2
```

Bitte beachten Sie, daß zwischen dem Dollarzeichen und dem Rest des Metabefehls *kein Leerzeichen* stehen darf.

Um Nur-Kommentar-Metabefehle in Kommentare zu setzen, geben Sie vor dem ersten Dollarzeichen in der Zeile ein Zeichen ein, das kein Tabulator oder Leerzeichen ist. Beispielsweise werden in der folgenden Zeile beide Metabefehle übergangen:

```
REM x$METABEFEHL1  $METABEFEHL2
```

C.2 Wie Sie zusätzliche Quelldateien verarbeiten: $INCLUDE

Der Metabefehl **$INCLUDE** weist den Compiler an, vorübergehend die Verarbeitung einer Datei zu verlassen und stattdessen Programmanweisungen der im Argument angegebenen BASIC-Datei zu lesen. Wenn das Ende der eingebundenen Datei erreicht ist, setzt der Compiler die Verarbeitung der ursprünglichen Datei fort. Da die Kompilierung mit der Zeile beginnt, die unmittelbar auf die Zeile mit dem Befehl **$INCLUDE** folgt, sollte **$INCLUDE** die letzte Anweisung in einer Zeile sein. Die nachstehende Anweisung ist richtig:

```
DEFINT I-N     $INCLUDE:  'COMMON.BAS'
```

Die beiden folgenden Einschränkungen sind bei der Benutzung von Include-Dateien zu beachten:

1. Eingebundene Dateien dürfen keine **SUB**- oder **FUNCTION**-Anweisungen enthalten.
2. Eingebundene Dateien, die mit BASICA erstellt wurden, müssen mit der Option **,A** gespeichert werden.

C.3 Zuordnung von dimensionierten Datenfeldern: $STATIC und $DYNAMIC

Die **$STATIC**- und **$DYNAMIC**-Metabefehle teilen dem Compiler mit, wie Speicher für Datenfelder zugeordnet werden soll.

$STATIC reserviert Speicher für Datenfelder während der Kompilierung. Wenn **$STATIC** benutzt wird, reinitialisiert die **ERASE**-Anweisung alle Datenfeldwerte mit Null (numerische Datenfelder) oder mit der Null-Zeichenkette (Zeichenketten-Datenfelder). Hierbei werden die Datenfelder nicht entfernt. **REDIM** hat keine Auswirkungen auf **$STATIC**-Datenfelder.

$DYNAMIC ordnet Datenfeldern Speicherplatz während des Programmlaufs zu. Das bedeutet, daß die **ERASE**-Anweisung die Datenfelder entfernt und den Speicherplatz, den sie in Anspruch genommen haben, für andere Zwecke freigibt. Sie können auch die **REDIM**-Anweisung zur Veränderung der Größe eines **$DYNAMIC**-Datenfeldes benutzen.

Die **$STATIC**- und **$DYNAMIC**-Metabefehle haben Auswirkungen auf alle Datenfelder, jedoch nicht auf implizit dimensionierte Datenfelder (Datenfelder, die nicht in einer **DIM**-Anweisung deklariert sind). Implizit dimensionierte Datenfelder werden stets so zugeordnet, als wäre **$STATIC** benutzt worden.

In Kapitel 2, "Datentypen", finden Sie weitere Informationen zu der Verwendung von **$STATIC** und **$DYNAMIC**.



Anhang D Fehlermeldungen

Während der Entwicklung eines BASIC-Programmes mit QuickBASIC können die folgenden Fehlerarten auftreten:

- Aufruffehler
- Kompilierzeitfehler
- Kompilierzeitwarnungen
- Linkzeitfehler
- Laufzeitfehler

Jeder Fehlertyp ist mit einem bestimmten Schritt im Entwicklungsprozeß eines Programmes verbunden. Aufruffehler treten während des Aufrufens von QuickBASIC mit den Befehlen **qb** oder **bc** auf. Kompilierzeitfehler und -warnungen treten während der Kompilierung, Laufzeitfehler während der Ausführung des Programmes auf. Linkzeitfehler treten nur auf, wenn Sie den Befehl **link** verwenden, um Objektdateien zu binden, die mit **bc** oder anderen Sprach-Compilern erstellt wurden.

Abschnitt D.1 listet die Aufruf-, Kompilierzeit- und Laufzeitfehlermeldungen alphabetisch, zusammen mit den jeweils zugewiesenen Fehlercodes, auf. In der Tabelle auf der nächsten Seite werden die Laufzeitfehlermeldungen und -fehlercodes in numerischer Reihenfolge aufgelistet. Abschnitt D.2 listet die Fehlermeldungen des Microsoft Overlay-Linkers und Abschnitt D.3 die Fehlermeldungen des Microsoft-Bibliotheksmanagers auf.

Wenn Laufzeitfehler in Programmen auftreten, die innerhalb der QuickBASIC-Umgebung gestartet wurden, erscheint eine Fehlermeldung. Die Zeile, in der der Fehler auftritt, wird auf eine der folgenden Arten angezeigt:

1. Auf einem Farbmonitor mit einem Farbgrafikadapter wird die Zeile grün dargestellt.
2. Wenn QuickBASIC mit der Option **/b** aufgerufen wurde, wird die Zeile auf einem Farbmonitor mit intensiverer Helligkeit dargestellt.
3. Auf einem monochromen Monitor wird die Zeile unterstrichen.

In selbständig ausführbaren Programmen (d. h. Programmen, die durch Eingabe des Basisnamens der ausführbaren Datei bei der Systemanfrage ausgeführt werden), gibt das Laufzeitsystem die Fehlermeldungen aus, gefolgt von einer Adresse, sofern nicht eine der Optionen **/d**, **/e** oder **/w** in der **bc**-Befehlszeile angegeben wird. In diesen Fällen folgt der Fehlermeldung außerdem die Nummer der Zeile, in der der Fehler aufgetreten ist. Die Standardformate dieses Typs von Fehlermeldungen sind die folgenden:

`Fehler` *n* `in Modul` *Modulname* `bei Adresse` *Segment:Offset*

und

`Fehler` *n* `in Zeile` *Zeilennummer* `von Modul` *Modulname* `bei Adresse` *Segment:Offset.*

Bei einigen Fehlern wird ein "**ERR**-Code" aufgelistet. Tritt ein Fehler auf, so wird die **ERR**-Variable auf den entsprechenden Code gesetzt, wenn eine Fehlerverfolgungs-Unterroutine aufgerufen wird. (Fehlerverfolgungs-Routinen werden über die Anweisung **ON ERROR** aufgerufen.) Die **ERR**-Variable behält diesen Wert, bis eine **RESUME**-Anweisung die Kontrolle an das Hauptprogramm zurückgibt. Weitere Informationen hierzu finden Sie in Kapitel 6, "Fehler- und Ereignisverfolgung", in *Programmieren in BASIC: Ausgewählte Themen.*

Die folgende Tabelle listet die Fehlercodes in numerischer Reihenfolge auf. Erläuterungen zu den Fehlern finden Sie in der alphabetischen Auflistung.

Laufzeifehlermeldungen

3	RETURN ohne GOSUB	54	Falscher Dateimodus
4	Außerhalb von DATA	55	Datei bereits geöffnet
5	Unzulässiger Funktionsaufruf	56	FIELD-Anweisung aktiv
6	Überlauf	57	Geräte-E/A-Fehler
7	Speicherkapazität reicht nicht aus	58	Datei existiert bereits
9	Index außerhalb des Bereichs	59	Falsche Satzlänge
11	Division durch Null	61	Diskette/Festplatte voll
14	Platz für Zeichenkette nicht ausreichend	62	Eingabe nach Dateiende
16	Zeichenkettenformel zu umfangreich	63	Falsche Datensatznummer
19	RESUME fehlt	64	Unzulässiger Dateiname
20	RESUME ohne Fehler	67	Zu viele Dateien
24	Laufzeitfehler am Gerät	68	Gerät nicht verfügbar
25	Gerätefehler	69	Überlauf des Kommunikationspuffers
27	Papier zu Ende	70	Zugriff nicht gestattet
39	CASE ELSE erwartet	71	Diskette/Festplatte nicht bereit
40	Variable erforderlich	72	Datenträgerfehler
50	FIELD-Überlauf	73	Erweiterte Eigenschaft nicht verfügbar
51	Interner Fehler	74	Umbenennen zwischen Disketten/Festplatte
52	Falscher Dateiname oder falsche Dateinummer	75	Pfad/Datei-Zugriffsfehler
53	Datei nicht gefunden	76	Pfad nicht gefunden

D.1 Aufruf-, Kompilierzeit- und Laufzeitfehlermeldungen

`"FN" am Beginn nicht möglich`

Sie haben "FN" als die ersten beiden Buchstaben eines Unterprogramm- oder Variablennamens benutzt. "FN" kann nur für die ersten beiden Buchstaben verwendet werden, wenn eine **DEF FN**-Funktion aufgerufen wird. (Kompilierzeitfehler)

`$INCLUDE-Datei-Zugriffsfehler`

Die im Metabefehl **$INCLUDE** genannte Include-Datei kann nicht gefunden werden. (Kompilierzeitfehler)

`$Metabefehl-Fehler`

Ein Metabefehl ist falsch. Wenn Sie das Programm mit **bc** kompilieren, ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`/C: Puffergröße zu groß`

Die maximale Größe des Kommunikationspuffers beträgt 32.767 Bytes. (**bc**-Aufruffehler)

`ALIAS erfordert Zeichenkettenkonstante`

Das Schlüsselwort **ALIAS** der Anweisung **DECLARE** erfordert als Argument eine Zeichenkettenkonstante. (Kompilierzeitfehler)

`Angabe eines unzulässigen formalen Parameters`

Es liegt ein Fehler in der Parameterliste einer Funktion oder eines Unterprogramms vor. (Kompilierzeitfehler)

`Anweisung ignoriert`

Sie verwenden den **bc**-Befehl zum Kompilieren eines Programmes, das **TRON**- und **TROFF**-Anweisungen enthält, ohne die Option **/d** zu benutzen. Dieser Fehler ist nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Anweisung in $INCLUDE-Datei unzulässig`

SUB...END SUB- und **FUNCTION...END FUNCTION**-Anweisungsblöcke sind in Include-Dateien nicht zulässig. Benutzen Sie den Befehl **Zusammenführen** aus dem Menü **Datei**, um die Include-Datei in das aktuelle Modul einzufügen, oder laden Sie die Include-Datei als ein separates Modul. Wenn Sie die Include-Datei als ein separates Modul laden, können einige Umstrukturierungen notwendig werden, weil gemeinsam benutzte Variablen nur innerhalb des Bereiches eines Moduls geteilt werden können. (Kompilierzeitfehler)

`Anweisung in TYPE-Block unzulässig`

Die einzigen Anweisungen, die zwischen den Anweisungen **TYPE**- und **END TYPE** stehen dürfen, sind **REM** und *Element* **AS** *Typname*. (Kompilierzeitfehler)

`Anweisung kann nicht vor SUB/FUNCTION-Definition stehen`

Die einzigen Anweisungen, die vor einer Prozedurdefinition stehen dürfen, sind **REM** und **DEF***Typ*. (Kompilierzeitfehler)

`Anweisung muß am Zeilenanfang stehen`

In Block-**IF...THEN...ELSE**-Konstruktionen darf vor **IF**, **ELSE**, **ELSEIF** und **END IF** nur eine Zeilennummer oder eine Zeilenmarke stehen. (Kompilierzeitfehler)

`Anweisungen/Marken zwischen SELECT CASE und CASE unzulässig`

Anweisungen und Zeilenmarken sind zwischen **SELECT CASE** und der ersten **CASE**-Anweisung nicht zulässig. Erlaubt sind Kommentare und Anweisungstrenner. (Kompilierzeitfehler)

`AS-Klausel bei erster Deklaration erforderlich`

Auf eine Variable, die nicht unter Verwendung einer **AS**-Klausel deklariert ist, wird in einer **AS**-Klausel Bezug genommen. (Kompilierzeitfehler)

`AS-Klausel erforderlich`

Es wird auf eine Variable, die mit einer **AS**-Klausel deklariert ist, ohne diese Klausel Bezug benommen. Falls die erste Deklaration einer Variablen eine **AS**-Klausel hat, muß jede folgende **DIM, REDIM, SHARED** und **COMMON**-Anweisung, die Bezug auf diese Variable nimmt, eine **AS**-Klausel enthalten. (Kompilierzeitfehler)

`Ausdruck zu komplex`

Dieser Fehler wird verursacht, wenn bestimmte interne Begrenzungen überschritten werden. Zum Beispiel: Während der Auswertung eines Ausdrucks werden Zeichenketten, die nicht mit Variablen verknüpft sind, temporäre Adressen zugewiesen. Durch eine große Anzahl solcher Zeichenketten kann dieser Fehler auftreten. Versuchen Sie, die Ausdrücke zu vereinfachen und Zeichenketten Variablen zuzuweisen. (Kompilierzeitfehler)

`Außerhalb des Datenbereiches`

Versuchen Sie, den erforderlichen Datenbereich wie folgt zu verändern:

- Verwenden Sie in der **LEN**-Klausel der Anweisung **OPEN** einen kleineren Dateipuffer.
- Verwenden Sie für das Anlegen dynamischer Datenfelder den Metabefehl **$DYNAMIC**. Daten dynamischer Datenfelder können normalerweise viel größer als Daten statischer Datenfelder sein.
- Verwenden Sie anstelle von Datenfeldern mit Zeichenketten variabler Länge Datenfelder mit Zeichenketten fester Länge.
- Verwenden Sie den kleinsten Datentyp, der Ihrer Aufgabe gerecht wird. Verwenden Sie Ganzzahlen wo immer dies möglich ist.
- Versuchen Sie nicht, viele kleine Prozeduren zu verwenden. QuickBASIC muß zu jeder Prozedur einige Bytes Kontrollinformationen erstellen.
- Verwenden Sie **CLEAR**, um die Größe des Stapels (Stack) zu verändern. Verwenden Sie nicht mehr Stapelplatz, als Ihre Aufgabe erfordert.
- Verwenden Sie keine Quellzeilen, die länger als 256 Zeichen lang sind. Solche Zeilen erfordern die Zuweisung von zusätzlichem Textpufferbereich.

(Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 7

`Außerhalb des Stapelbereiches`

Dieser Fehler kann auftreten, wenn eine rekursive **FUNCTION** zu tief verschachtelt ist oder zu viele aktive Aufrufe von Unterroutinen, **SUB**'s und **FUNCTION**'s existieren. Um den dem Programm zugewiesenen Stapelplatz zu vergrößern, können Sie die **CLEAR**-Anweisung verwenden. Dieser Fehler kann nicht verfolgt werden. (Laufzeitfehler)

`Außerhalb von DATA`

Es wird eine **READ**-Anweisung ausgeführt, obwohl in dem Programm keine **DATA**-Anweisungen mit ungelesenen Daten übrig sind. (Laufzeitfehler)

ERR-Code: 4

`Benutzerdefinierter Variablentyp in Ausdruck unzulässig`

Variablen mit benutzerdefiniertem Typ sind in Ausdrücken wie `CALL ALPHA ((X))` nicht zulässig, wobei `X` ein benutzerdefinierter Typ ist. (Kompilierzeitfehler)

`Bezeichner dürfen keinen Punkt enthalten`

Bezeichner von benutzerdefinierten Typen und Namen von Verbundelementen dürfen keine Punkte enthalten. Der Punkt sollte nur als Verbundvariablentrenner benutzt werden. Zusätzlich darf ein Variablenname keinen Punkt enthalten, wenn der Teil des Namens vor dem Punkt irgendwo im Programm in einer Klausel *Bezeichner* **AS** *Benutzertyp* verwendet wird. Wenn Sie ein Programm benutzen, das den Punkt in Variablennamen verwendet, ist es ratsam, daß Sie diesen durch Groß-/Kleinschreibung ersetzen. Aus der Variablen "**ALPHA.BETA**" beispielsweise würde "AlphaBeta" werden. (Kompilierzeitfehler)

`Bezeichner erwartet`

Sie versuchen, eine Zahl oder ein BASIC-reserviertes Wort zu benutzen, während ein Bezeichner erwartet wird. (Kompilierzeitfehler)

`Bezeichner kann nicht mit %, &, !, # oder $ enden`

Die oben aufgeführten Zusätze sind nicht in Typbezeichnern, Unterprogrammnamen oder benannten **COMMON**-Namen erlaubt. (Kompilierzeitfehler)

`Bezeichner zu lang`

Bezeichner dürfen nicht länger als 40 Zeichen sein. (Kompilierzeitfehler)

`Binäre Quelldatei`

Die Datei, die Sie zu kompilieren versucht haben, ist keine ASCII-Datei. Alle von BASICA gespeicherten Quelldateien sollten mit der Option **,A** gespeichert werden. (Kompilierzeitfehler)

`Block-IF ohne END IF`

In einer Block-**IF**-Konstruktion fehlt das zugehörige **END IF**. (Kompilierzeitfehler)

`BYVAL gestattet nur numerische Argumente`

BYVAL akzeptiert keine Zeichenketten- oder Verbundargumente. (Kompilierzeitfehler)

`CASE ELSE erwartet`

Für einen Ausdruck in einer **SELECT CASE**-Anweisung wurde kein passender Fall gefunden. (Laufzeitfehler)

ERR-Code: 39

`CASE ohne SELECT`

Der erste Teil einer **SELECT CASE**-Anweisung fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`COMMON in Quick-Bibliothek zu klein`

In dem Modul sind mehr gemeinsame Variablen angegeben als in der aktuell geladenen Quick-Bibliothek. (Kompilierzeitfehler)

`COMMON und DECLARE müssen vor ausführbaren Anweisungen stehen`

Eine **COMMON**- oder **DECLARE**-Anweisung ist falsch plaziert. **COMMON**- und **DECLARE**-Anweisungen müssen vor der ersten ausführbaren Anweisung stehen. Alle BASIC-Anweisungen, bis auf die folgenden, sind ausführbar:

- **COMMON**
- **DEF***Typ*
- **DIM** (für statische Datenfelder)
- **OPTION BASE**
- **REM**
- **TYPE**
- Alle Metabefehle

(Kompilierzeitfehler)

`CONST/DIM SHARED folgt auf SUB/FUNCTION`

Die Anweisungen **CONST** und **DIM SHARED** sollten vor irgendeiner Definition eines Unterprogrammes oder **FUNCTION**-Prozedur erscheinen. Wenn Sie Ihr Programm mit **bc** kompilieren, ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Ergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Datei` (*Dateiname*) `nicht gefunden. Pfad eingeben:`

Dieser Fehler tritt auf, wenn QuickBASIC eine Quick-Bibliothek oder ein Quick-Dienstprogramm (**BC.EXE**, **LINK.EXE**, **LIB.EXE** oder **QB.EXE**), die/das vom Programm benötigt wird, nicht finden kann. Geben Sie den Pfad-Namen korrekt ein oder betätigen Sie STRG+C, um zur DOS-Anfrage zurückzukehren. (**qb**-Aufruffehler)

`Datei bereits geöffnet`

Für eine bereits geöffnete Datei wird ein sequentieller Ausgabemodus in OPEN angegeben, oder für eine geöffnete Datei wird **KILL** angegeben. (Laufzeitfehler)

ERR-Code: 55

`Datei existiert bereits`

Der in einer **NAME**-Anweisung angegebene Dateiname ist mit einem Dateinamen, der bereits auf der Diskette/Festplatte verwendet wird, identisch. (Laufzeitfehler)

ERR-Code: 58

`Datei ist bereits geladen`

Sie versuchen, eine Datei zu laden, die sich bereits im Speicher befindet. (Kompilierzeitfehler)

`Datei nicht gefunden`

Eine Anweisung **KILL, NAME, FILES** oder **OPEN** bezieht sich auf eine Datei, die an der spezifizierten Stelle nicht existiert. (Laufzeitfehler)

ERR-Code: 53

`Datenfeld bereits dimensioniert`

Dieser Fehler kann verursacht werden durch:

- Mehr als eine **DIM**-Anweisung für dasselbe statische Datenfeld.
- Eine **DIM**-Anweisung nach erster Verwendung eines Datenfeldes. Die Zuordnung für statische Datenfelder muß mit der Anweisung **ERASE** aufgehoben werden, bevor die Felder neu dimensioniert werden können; dynamische Datenfelder können ebenso mit der Anweisung **REDIM** erneut dimensioniert werden.
- Eine **OPTION BASE**-Anweisung erscheint, nachdem ein Datenfeld dimensioniert worden ist.

(Kompilierzeit- oder Laufzeitfehler)

`Datenfeld ist nicht definiert`

Es wird auf ein Datenfeld Bezug genommen, das nicht definiert ist. (Kompilierzeitfehler)

`Datenfeld ist nicht dimensioniert`

Es wird auf ein Datenfeld Bezug genommen, das nicht dimensioniert ist. Wenn Sie das Programm mit **bc** kompilieren, ist dieser Fehler nicht "fatal"; das Programm wird zwar ausgeführt, die Programmergebnisse können aber falsch sein. (Kompilierzeitwarnung)

`Datenfeld zu groß`

Der Platz für Benutzerdaten reicht für die Aufnahme des Feldes nicht aus. Verkleinern Sie das Datenfeld oder benutzen Sie den **$DYNAMIC**-Metabefehl. Sie können diesen Fehler auch erhalten, wenn die Größe des Datenfeldes 64K überschreitet, das Datenfeld nicht dynamisch ist und die **/ah**-Option nicht verwendet wurde. Reduzieren Sie die Größe des Datenfeldes oder machen Sie das Datenfeld dynamisch und benutzen die **/ah**-Befehlszeilenoption. (Kompilierzeitfehler)

`Datenträgerfehler`

Die Hardware des Disketten-/Festplattenlaufwerks hat einen physischen Defekt auf der Diskette/Festplatte entdeckt. (Laufzeitfehler)

ERR-Code: 72

`DECLARE erforderlich`

Ein impliziter Aufruf eines Unterprogramms oder einer **FUNCTION** erscheint vor der Definition der Prozedur. (Ein impliziter Aufruf verwendet nicht die Anweisung **CALL**.) Alle Prozeduren müssen definiert oder deklariert sein, bevor sie implizit aufgerufen werden. (Kompilierzeitfehler)

`DEF FN in Steuerungsanweisungen nicht erlaubt`

Definitionen von **DEF FN**-Funktionen sind innerhalb von Steuerungsstrukturen wie **IF...THEN...ELSE** und **SELECT CASE** nicht erlaubt. (Kompilierzeitfehler)

`DEF ohne END DEF`

In einer mehrzeiligen Funktionsdefinition gibt es kein zugehöriges **END DEF**. (Kompilierzeitfehler)

`DEF`*Typ*`-Zeichenangabe unzulässig`

Eine **DEF***Typ*-Anweisung ist falsch eingegeben. Auf **DEF** kann nur **DBL**, **INT**, **LNG**, **SNG**, **STR** oder (bei vom Benutzer definierten Funktionen) ein Leerzeichen folgen. (Kompilierzeitfehler)

`Diskette nicht bereit`

Das Laufwerk ist nicht verriegelt, oder es befindet sich keine Diskette im Laufwerk.

ERR-Code: 71

`Diskette/Festplatte voll`

Es ist nicht genügend Platz auf der Diskette/Festplatte, um eine **PRINT**-, **WRITE**- oder **CLOSE**-Operation abzuschließen. Dieser Fehler kann ebenfalls auftreten, wenn für den QuickBASIC nicht genügend Platz vorhanden ist, eine **.OBJ**- oder **.EXE**-Datei zu erzeugen. (Laufzeitfehler)

ERR-Code: 61

`Division durch Null`

In einem Ausdruck wird eine Division durch Null vorgenommen oder Null wird mit einem negativen Wert potenziert. Dieser Fehler tritt ebenfalls auf, wenn die Ganzzahl -32.768 durch 1 oder -1 dividiert wird, oder bei der Modulo-Division von -32.768 durch 1 oder durch -1. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 11

`DO ohne LOOP`

In einer **DO...LOOP**-Anweisung fehlt die abschließende **LOOP**-Klausel. (Kompilierzeitfehler)

`Dokument zu groß`

Ihr Dokument überschreitet die interne Grenze von QuickBASIC. Teilen Sie das Dokument in separate Dateien auf.

`Doppelpunkt nach /C erwartet`

Zwischen der Option und dem Puffergrößenargument wird ein Doppelpunkt verlangt. (**bc**-Aufruffehler)

`Doppelte Definition`

Sie benutzen einen Bezeichner, der bereits definiert wurde. Sie versuchen beispielsweise, denselben Namen sowohl in einer **CONST**-Anweisung als auch als Variablendefinition zu verwenden.

Dieser Fehler tritt ebenfalls auf, wenn Sie versuchen, ein Datenfeld erneut zu dimensionieren. Für die erneute Dimensionierung dynamischer Datenfelder müssen Sie **DIM** oder **REDIM** verwenden. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 10

`Doppelte Marke`

Zwei Programmzeilen wird dieselbe Nummer oder Marke zugewiesen. Jede Zeilennummer oder Zeilenmarke darf in einem Modul nur einmal vorkommen. (Kompilierzeitfehler)

`DOS 2.10 oder höhere Version erforderlich`

Sie versuchen, QuickBASIC mit einer falschen DOS-Version zu verwenden. (**qb**-Aufruf- oder Laufzeitfehler)

`Dynamisches Datenfeldelement unzulässig`

Dynamische Datenfeldelemente sind mit **VARPTR$** nicht erlaubt. (Kompilierzeitfehler)

`Einfache Variable oder Datenfeldvariable erwartet`

QuickBASIC erwartet ein Variablenargument. (Kompilierzeitfehler)

`Eingabe nach Dateiende`

Eine **INPUT**-Anweisung liest aus einer leeren oder aus einer Datei ein, aus der sämtliche Daten bereits gelesen sind. Verwenden Sie zur Vermeidung dieses Fehlers die Funktion **EOF**, um das Dateiendezeichen zu ermitteln. (Laufzeitfehler)

ERR-Code: 62

`Eingegebene Datei nicht gefunden`

Die Quelldatei, die Sie auf der Befehlszeile angegeben haben, befindet sich nicht am angegebenen Ort. (**bc**-Aufruffehler)

`Element nicht definiert`

Es wird auf ein Element eines benutzerdefinierten Typs Bezug genommen, ohne daß es definiert ist. Wenn beispielsweise der benutzerdefinierte Typ `MEINTYP` die Elemente `A`, `B` und `C` enthält, würde der Versuch, die Variable `D` als ein Element von `MEINTYP` zu verwenden, zu dieser Fehlermeldung führen. (Kompilierzeitfehler)

`ELSE ohne IF`

Eine **ELSE**-Klausel erscheint ohne ein dazugehöriges **IF**. Manchmal entsteht dieser Fehler durch falsches Verschachteln von **IF**-Anweisungen. (Kompilierzeitfehler)

`ELSEIF ohne IF`

Eine **ELSEIF**-Anweisung erscheint ohne ein dazugehöriges **IF**. Manchmal entsteht dieser Fehler durch falsches Verschachteln von **IF**-Anweisungen. (Kompilierzeitfehler)

`END DEF ohne DEF`

Eine Anweisung **END DEF** hat keine zugehörige Anweisung **DEF**. (Kompilierzeitfehler)

`END IF ohne Block-IF`

Der Beginn eines **IF**-Blockes fehlt. (Kompilierzeitfehler)

`END SELECT ohne SELECT`

Das Ende einer **SELECT CASE**-Anweisung erscheint ohne ein beginnendes **SELECT CASE**. Der Beginn der **SELECT CASE**-Anweisung könnte fehlen oder falsch geschrieben sein. (Kompilierzeitfehler)

`END SUB oder END FUNCTION muß letzte Zeile im Fenster sein`

Sie versuchen, Code nach einer Prozedur anzufügen. Sie müssen entweder zum Hauptmodul zurückkehren oder ein anderes Modul öffnen. (Kompilierzeitfehler)

`END SUB/FUNCTION ohne SUB/FUNCTION`

Sie haben die **SUB**- oder **FUNCTION**-Anweisung gelöscht. (Kompilierzeitfehler)

`END TYPE ohne TYPE`

Eine **END TYPE**-Anweisung wird außerhalb einer **TYPE**-Deklaration benutzt. (Kompilierzeitfehler)

`Erwartet:` *Objekt*

Dies ist ein Syntaxfehler. Der Cursor ist auf dem nicht erwarteten Objekt positioniert. (Kompilierzeitfehler)

`Erweiterte Eigenschaft nicht verfügbar`

Sie versuchen eine Eigenschaft von QuickBASIC zu verwenden, die mit einer anderen BASIC-Version verfügbar ist. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 73

`EXIT befindet sich nicht innerhalb von FOR...NEXT`

Eine **EXIT FOR**-Anweisung wird außerhalb einer **FOR...NEXT**-Anweisung benutzt. (Kompilierzeitfehler)

`EXIT DO befindet sich nicht innerhalb von DO...LOOP`

Eine **EXIT DO**-Anweisung wird außerhalb einer **DO...LOOP**-Anweisung benutzt. (Kompilierzeitfehler)

`Falsche Anzahl von Dimensionen`

Der Bezug auf ein Datenfeld enthält die falsche Anzahl von Dimensionen. (Kompilierzeitfehler)

`Falsche Datensatznummer`

In einer **PUT**- oder **GET**-Anweisung ist die Nummer des Datensatzes kleiner gleich Null. (Laufzeitfehler)

ERR-Code: 63

`Falsche Satzlänge`

Es wird eine **GET**- oder **PUT**-Anweisung ausgeführt, die eine Verbundvariable angibt, deren Länge nicht zur angegebenen Satzlänge in der zugehörigen **OPEN**-Anweisung paßt. (Laufzeitfehler)

ERR-Code: 59

`Falscher Dateimodus`

Dieser Fehler tritt in den folgenden Situationen auf:

1. Das Programm versucht, **PUT** oder **GET** für eine sequentielle Datei zu verwenden, oder ein **OPEN** mit einem anderen Modus als **I**, **O** oder **R** auszuführen.
2. Das Programm versucht, eine **FIELD**-Anweisung für eine Datei zu verwenden, die nicht für Direktzugriff geöffnet ist.
3. Das Programm versucht, in eine für Eingabe geöffnete Datei zu schreiben.
4. Das Programm versucht, aus einer für Ausgabe oder Erweiterung geöffneten Datei zu lesen.
5. QuickBASIC versucht, eine Include-Datei, die im komprimierten Format gespeichert wurde, zu verwenden. Include-Dateien müssen im Textformat gespeichert werden. Laden Sie die Include-Datei erneut, speichern Sie sie im Textformat, und versuchen Sie, das Programm erneut zu starten.

(Laufzeitfehler)

ERR-Code: 54

`Falscher Dateiname oder falsche Dateinummer`

Eine Anweisung oder ein Befehl bezieht sich auf eine Datei mit einem Dateinamen oder einer Dateinummer, die nicht in der **OPEN**-Anweisung angegeben wurde oder die sich außerhalb des bei der Initialisierung angegebenen Bereiches von Dateinummern befindet. (Laufzeitfehler)

ERR-Code: 52

`Fehlende linke Klammer`

QuickBASIC erwartet eine linke Klammer. (Kompilierzeitfehler)

`Fehlende Option On Error (/E)`

Wenn Sie mit dem **bc**-Befehl kompilieren, müssen Programme, die **ON ERROR GOTO**-Anweisungen enthalten, mit der Option On Error (/**e**) kompiliert werden. (Kompilierzeitfehler)

`Fehlende Option Resume Next (/X)`

Wenn Sie mit dem **bc**-Befehl kompilieren, müssen Programme, die die Anweisungen **RESUME, RESUME NEXT** oder **RESUME 0** enthalten, mit der Option Resume Next (/**x**) kompiliert werden. (Kompilierzeitfehler)

`Fehlende rechte Klammer`

QuickBASIC erwartet eine rechte (schließende) Klammer. (Kompilierzeitfehler)

`Fehlende Zeilennummer oder Zeilenmarke`

Eine Zeilennummer oder Zeilenmarke fehlt bei einer Anweisung, die dies erfordert, z. B. **GOTO**. (Kompilierzeitfehler)

`Fehlender Stern`

In einer Zeichenkettendefinition eines benutzerdefinierten Typs fehlt der Stern. (Kompilierzeitfehler)

`Fehlendes AS`

QuickBASIC erwartet ein **AS**-Schlüsselwort wie in `OPEN "DATEINAME" FOR INPUT AS #1`. (Kompilierzeitfehler)

`Fehlendes BASE`

QuickBASIC hat hier das Schlüsselwort **BASE**, wie in **OPTION BASE**, erwartet. (Kompilierzeitfehler)

`Fehlendes Gleichheitszeichen`

QuickBASIC erwartet ein Gleichheitszeichen. (Kompilierzeitfehler)

`Fehlendes GOSUB`

Einer **ON** *Ereignis*-Anweisung fehlt das **GOSUB**. (Kompilierzeitfehler)

`Fehlendes GOTO`

Einer **ON ERROR**-Anweisung fehlt das **GOTO**. (Kompilierzeitfehler)

`Fehlendes INPUT`

QuickBASIC erwartet das Schlüsselwort **INPUT**. (Kompilierzeitfehler)

`Fehlendes Komma`

QuickBASIC erwartet ein Komma. (Kompilierzeitfehler)

`Fehlendes Minuszeichen`

QuickBASIC erwartet ein Minuszeichen. (Kompilierzeitfehler)

`Fehlendes Semikolon`

QuickBASIC erwartet ein Semikolon. (Kompilierzeitfehler)

`Fehlende SUB oder FUNCTION`

Eine **DECLARE**-Anweisung hat keine zugehörige Prozedur. (Kompilierzeitfehler)

`Fehlendes THEN`

QuickBASIC erwartet das Schlüsselwort **THEN**. (Kompilierzeitfehler)

`Fehlendes TO`

QuickBASIC erwartet das Schlüsselwort **TO**. (Kompilierzeitfehler)

`Fehlendes TYPE`

Einer **END TYPE**-Anweisung fehlt das Schlüsselwort **TYPE**. (Kompilierzeitfehler)

`Fehler beim Laden der Datei` (*Datei*)`—Datei nicht gefunden`

Dieser Fehler tritt bei der Umleitung von Eingaben in QuickBASIC aus einer Datei auf. Die Eingabedatei befindet sich nicht an der Stelle, die in der Befehlszeile angegeben wurde. (**qb**-Aufruffehler)

`Fehler beim Laden der Datei` (*Datei*)`—Disketten-E/A-Fehler`

Dieser Fehler wird von physikalischen Problemen bei Diskettenzugriffen verursacht, z. B. wenn das Laufwerk, das *Datei* enthält, nicht verriegelt ist. (**qb**-Aufruffehler)

`Fehler beim Laden der Datei` (*Datei*)`—DOS-Speicherbereichsfehler`

In den von DOS verwendeten Speicherbereich wurde entweder von einer Assembler-Routine oder mit einer **POKE**-Anweisung geschrieben. (**qb**-Aufruffehler)

`Fehler beim Laden der Datei` (*Datei*)`— Speicherplatz zu gering`

Es wird mehr Speicherplatz verlangt als verfügbar ist. Es kann zum Beispiel sein, daß nicht genügend Speicher für die Zuordnung eines Dateipuffers vorhanden ist. Versuchen Sie, die Größe Ihrer DOS-Puffer zu reduzieren, alle speicherresidenten Programme oder einige Gerätetreiber zu entfernen. Wenn Sie große Datenfelder haben, versuchen Sie, zu Beginn Ihres Programmes einen **$DYNAMIC**-Metabefehl einzufügen. Wenn Sie Dokumente geladen haben, wird deren Entfernung einigen Speicherplatz freigeben. (Laufzeitfehler)

ERR-Code: 7

`Fehler beim Laden der Datei` (*Datei*)`—ungültiges Format`

Sie versuchen, eine Quick-Bibliothek zu laden, die nicht das korrekte Format hat. Dieser Fehler kann auftreten, wenn Sie versuchen, eine mit einer früheren Version von QuickBASIC erstellte Quick-Bibliothek zu verwenden; wenn Sie versuchen, eine Datei zu verwenden, die nicht mit der QuickBASIC-Option **Bibliothek erstellen** oder der Linker-Option **/QU** verarbeitet wurde; oder wenn Sie versuchen, mit QuickBASIC eine **.LIB**-Bibliothek zu laden. (**qb**-Aufruffehler)

`Fehler während der QuickBASIC-Initialisierung`

Dieser Fehler kann verschiedene Ursachen haben. Am häufigsten tritt er auf, wenn die Maschine nicht genügend Speicherplatz hat, QuickBASIC zu laden. Falls Sie eine Benutzerbibliothek laden, versuchen Sie, die Bibliothek zu verkleinern.

Dieser Fehler tritt auf, wenn Sie versuchen, QuickBASIC mit nicht unterstützter Hardware zu verwenden. (**qb**-Aufruffehler)

`FIELD-Anweisung aktiv`

Es wurde eine **GET**- oder **PUT**-Anweisung ausgeführt, die eine Verbundvariable für eine Datei angegeben hat, für die ebenfalls **FIELD**-Anweisungen ausgeführt wurden. **GET** oder **PUT** mit einem Verbundvariablen-Argument dürfen nur für Dateien verwendet werden, für die keine **FIELD**-Anweisungen ausgeführt wurden. (Laufzeitfehler)

ERR-Code: 56

`FIELD-Überlauf`

Eine **FIELD**-Anweisung versucht, mehr Bytes zuzuweisen, als für die Satzlänge einer Direktzugriffsdatei angegeben wurden. (Laufzeitfehler)

ERR-Code: 50

`FOR ohne NEXT`

Jede **FOR**-Anweisung muß eine passende **NEXT**-Anweisung haben. (Kompilierzeitfehler)

`FOR-Schleifen-Indexvariable bereits belegt`

Dieser Fehler tritt auf, wenn eine Indexvariable in geschachtelten **FOR**-Schleifen mehr als einmal benutzt wird. (Kompilierzeitfehler)

`Formale Parameter nicht eindeutig`

Eine **FUNCTION**- oder Unterprogrammdeklaration enthält doppelte Parameter, wie in `SUB HolName (A, B, C, A) STATIC`. (Kompilierzeitfehler)

`Fortfahren nicht möglich`

Während des Debuggens haben Sie eine Änderung vorgenommen, die die weitere Ausführung verhindert. (Kompilierzeitfehler)

`Funktion bereits definiert`

Dieser Fehler tritt auf, wenn eine bereits definierte **FUNCTION** erneut definiert wird. (Kompilierzeitfehler)

`Funktion nicht definiert`

Sie müssen eine **FUNCTION** vor ihrer Verwendung deklarieren oder definieren. (Kompilierzeitfehler)

`Ganzzahl zwischen 1 und 32767 verlangt`

Die Anweisung verlangt ein ganzzahliges Argument. (Kompilierzeitfehler)

`Gerät nicht verfügbar`

Das Gerät, auf das Sie zuzugreifen versuchen, ist nicht on-line oder nicht vorhanden. (Laufzeitfehler)

ERR-Code: 68

`Geräte-E/A-Fehler`

Bei einer Geräte-E/A-Operation ist ein E/A-Fehler aufgetreten. Das Betriebssystem kann die Ausführung nach diesem Fehler nicht fortsetzen. (Laufzeitfehler)

ERR-Code: 57

`Gerätefehler`

Ein Gerät hat einen Hardwarefehler gemeldet. Wenn diese Meldung erfolgt, während Daten in eine Kommunikationsdatei übertragen werden, zeigt sie an, daß die mit der Anweisung **OPEN COM** geprüften Signale innerhalb des angegebenen Zeitraums nicht gefunden wurden. (Laufzeitfehler)

ERR-Code: 25

`GOTO oder GOSUB erwartet`

QuickBASIC erwartet eine **GOTO**- oder **GOSUB**-Anweisung. (Kompilierzeitfehler)

`Gültige Optionen: [RUN] Dat /AH /B /C:Puf /G /H /L [lib] /MBF /CMD ZeiKe`

Diese Meldung erscheint, wenn Sie QuickBASIC mit einer ungültigen Option aufrufen. (**qb**-Aufruffehler)

`In Prozedur oder DEF FN unzulässig`

Diese Anweisung ist innerhalb einer Prozedur nicht gestattet. (Kompilierzeitfehler)

`Include-Datei zu groß`

Ihre Include-Datei überschreitet die interne Grenze von QuickBASIC. Teilen Sie die Datei in separate Dateien. (Kompilierzeitfehler)

`Index außerhalb des Bereichs`

Auf einen Datensatz wurde mit einem Index, der außerhalb der Felddimensionen liegt, verwiesen, oder es wurde auf ein Element eines nicht dimensionierten dynamischen Datenfeldes zugegriffen. Diese Meldung erfolgt, wenn während des Kompilierens die Option Debug (**/d**) angegeben wurde. Sie können diesen Fehler auch erhalten, wenn die Größe des Datenfeldes 64K überschreitet, das Datenfeld nicht dynamisch ist und die **/ah**-Option nicht verwendet wurde. Reduzieren Sie die Größe des Datenfeldes oder machen Sie das Datenfeld dynamisch und benutzen Sie die **/ah**-Befehlszeilenoption. (Laufzeitfehler)

ERR-Code: 9

`Interner Fehler`

In QuickBASIC ist eine interne Störung aufgetreten.

ERR-Code: 51

`Interner Fehler bei` *xxxx*

In QuickBASIC ist bei Dateilage *xxxx* eine interne Störung aufgetreten.

`Kein Hauptmodul. Wählen Sie Hauptmodul bestimmen aus dem Menü Ausführen.`

Sie versuchen ein Programm zu starten, nachdem Sie das Hauptmodul entfernt haben. Jedes Programm muß ein Hauptmodul enthalten. (Kompilierzeitfehler)

`Keine Zeilennummer in` *Modulname* `bei Adresse` *Segment:Offset*

Dieser Fehler tritt auf, wenn während einer Fehlerverfolgung die Fehleradresse nicht in der Tabelle der Zeilennummern gefunden wird. Dies kommt vor, wenn keine ganzzahligen Zeilennummern zwischen 0 und 65.527 vorhanden sind. Dies kann ebenso auftreten, wenn das Benutzerprogramm versehentlich die Tabelle der Zeilennummern überschrieben hat. Dieser Fehler ist schwerwiegend und kann nicht verfolgt werden. (Laufzeitfehler)

`Korrigieren Sie die Eingabe`

Auf eine **INPUT**-Anfrage haben Sie mit der falschen Anzahl oder den falschen Typen von Objekten geantwortet. Geben Sie Ihre Antwort in der richtigen Form erneut ein. (Laufzeitfehler)

`Laufzeitfehler am Gerät`

Das Programm hat innerhalb einer vorher festgelegten Zeit keine Informationen von einem E/A-Gerät erhalten. (Laufzeitfehler)

ERR-Code: 24

`Lesefehler auf Standardeingabe`

Es tritt während des Lesens von der Konsole oder aus einer umgeleiteten Eingabedatei ein Systemfehler auf. (**bc**-Aufruffehler)

`Listing für binäre BASIC-Quelldatei kann nicht erstellt werden`

Sie versuchen, eine binäre Quelldatei mit dem Befehl **bc** und der Option **/a** zu kompilieren. Kompilieren Sie erneut ohne die Option **/a**. (**bc**-Aufruffehler)

`LOOP ohne DO`

Das eine **DO...LOOP**-Anweisung einleitende **DO** fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`Marke nicht definiert`

Es wird auf eine Zeilenmarke Bezug genommen (z. B. in einer **GOTO**-Anweisung), die in dem Programm nicht erscheint. (Kompilierzeitfehler)

`Marke nicht definiert:` *Marke*

Eine Anweisung **GOTO**-*Zeilenmarke* nimmt Bezug auf eine nicht existierende Zeilenmarke. (Kompilierzeitfehler)

`Mathematischer Überlauf`

Das Ergebnis einer Berechnung ist so groß, daß es im BASIC-Zahlenformat nicht dargestellt werden kann. (Kompilierzeitfehler)

`Modul nicht gefunden. Modul aus Programm entfernen?`

Beim Laden des Programmes hat QuickBASIC die Datei, die das angegebene Modul enthält, nicht gefunden. QuickBASIC hat statt dessen ein leeres Modul angelegt. Bevor Sie Ihr Programm starten können, müssen Sie dieses leere Modul löschen.

`Modul-Ebenen-Code zu groß`

Ihr Modul-Ebenen-Code überschreitet die internen Grenzen von QuickBASIC. Versuchen Sie, einen Teil des Codes in Unterprogramm- oder **FUNCTION**-Prozeduren unterzubringen. (Kompilierzeitfehler)

`Name des Unterprogramms unzulässig`

Dieser Fehler tritt auf, wenn ein Unterprogrammname ein in BASIC reserviertes Wort ist oder der Unterprogrammname zweimal vergeben wurde. (Kompilierzeitfehler)

`NEXT fehlt für` *Variable*

Einer **FOR**-Anweisung fehlt die zugehörige **NEXT**-Anweisung. Die *Variable* ist die Schleifen-Indexvariable des **FOR**. (Kompilierzeitfehler)

`NEXT ohne FOR`

Jede **NEXT**-Anweisung muß eine passende **FOR**-Anweisung haben. (Kompilierzeitfehler)

`Nicht anzeigbar`

Dieser Fehler tritt auf, wenn Sie in einem Anzeigeausdruck eine Variable angeben. Stellen Sie sicher, daß das Modul oder die Prozedur in dem aktiven Arbeitsbereich Zugriff auf die Variable hat, die Sie beobachten wollen. Zum Beispiel kann im Modul-Ebenen-Code nicht auf Variablen zugegriffen werden, die lokal zu einem Unterprogramm oder **FUNCTION** sind. (Laufzeitfehler)

`Nicht erkannter Schalterfehler: "QU"`

Sie versuchen, eine **.EXE**-Datei oder Quick-Bibliothek mit einer falschen Version des Microsoft Overlay-Linkers zu erstellen. Sie müssen den mit den Originaldisketten gelieferten Linker verwenden, um eine **.EXE**-Datei oder eine Quick-Bibliothek zu erstellen. (Kompilierzeitfehler)

`Nichtdruckbarer Fehler`

Für die vorliegende Fehlersituation gibt es keine Fehlermeldung. Ursache hierfür kann eine **ERROR**-Anweisung sein, die keinen definierten Fehlercode hat. (Laufzeitfehler)

`Numerisches Datenfeld unzulässig`

Numerische Datenfelder sind als Argumente für **VARPTR$** nicht zulässig. Nur einfache Variablen und Elemente von Zeichenkettendatenfeldern sind erlaubt. (Kompilierzeitfehler)

`Nur einfache Variablen erlaubt`

In **READ**- und **INPUT**-Anweisungen sind benutzerdefinierte Typen und Datenfelder nicht erlaubt. Datenfeldelemente, die nicht von einem benutzerdefinierten Typ sind, sind erlaubt. (Kompilierzeitfehler)

`Operation erfordert Diskette/Festplatte`

Sie versuchen auf bzw. von einem Nicht-Disketten-Gerät zu speichern bzw. zu laden, wie z. B. einem Drucker oder der Tastatur. (Kompilierzeitfehler)

`Option Ereignisver. (/W) oder Prüfen zw. Anweisungen (/V) fehlt`

Das Programm enthält eine **ON** *Ereignis*-Anweisung, die eine dieser Optionen benötigt. (Kompilierzeitfehler)

`Papier zu Ende`

Dem Drucker ist das Papier ausgegangen, oder er ist nicht eingeschaltet. (Laufzeitfehler)

ERR-Code: 27

`Pfad des Laufzeitmoduls eingeben:`

Diese Anfrage erscheint, wenn das Laufzeitmodul **BRUN40.EXE** nicht gefunden wird. Geben Sie den korrekten Pfad ein. Dies ist ein schwerwiegender Fehler, der nicht verfolgt werden kann. (Laufzeit-Anfrage)

`Pfad nicht gefunden.`

Während einer Operation mit **OPEN, MKDIR, CHDIR** oder **RMDIR** konnte DOS das angegebene Verzeichnis nicht finden. Die Operation wird nicht beendet. (Laufzeitfehler)

ERR-Code:76

`Pfad/Datei-Zugriffsfehler`

Während einer Operation mit **OPEN, MKDIR, CHDIR** oder **RMDIR** konnte das Betriebssystem keine korrekte Pfad/Dateinamen-Verbindung herstellen. Die Operation wird nicht beendet. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code:75

`Platz für Zeichenkette nicht ausreichend`

Zeichenkettenvariablen überschreiten den zugeordneten Zeichenkettenbereich. (Laufzeitfehler)

ERR-Code: 14

`Prozedur bereits in Quick-Bibliothek definiert`

Eine Prozedur in der Quick-Bibliothek hat denselben Namen wie eine Prozedur in Ihrem Programm. (Kompilierzeitfehler)

`Prozedur zu groß`

Diese Prozedur überschreitet die internen Grenzen von QuickBASIC. Verkleinern Sie die Prozedur durch Aufteilung in mehrere Prozeduren. (Kompilierzeitfehler)

`Puffergröße erwartet nach /C:`

Sie müssen eine Puffergröße nach der Option **/c** angeben. (**bc**-Aufruffehler)

`RESUME fehlt`

Während das Programm sich in einer Fehlerverfolgungsroutine befand, wurde das Programmende erreicht. Zur Behebung dieser Situation wird eine **RESUME**-Anweisung benötigt. (Laufzeitfehler)

ERR-Code: 19

`RESUME ohne Fehler`

Es wird eine **RESUME**-Anweisung erreicht, bevor eine Fehlerverfolgungsroutine aufgerufen wird. (Laufzeitfehler)

ERR-Code: 20

`RETURN ohne GOSUB`

Es wird eine **RETURN**-Anweisung erreicht, für die es keine vorhergehende passende **GOSUB**-Anweisung gibt. (Laufzeitfehler)

ERR-Code: 3

`SEG oder BYVAL in CALLS nicht erlaubt`

BYVAL und **SEG** sind nur in einer **CALL**-Anweisung zulässig. (Kompilierzeitfehler)

`SELECT ohne END SELECT`

Das Ende einer **SELECT CASE**-Anweisung fehlt oder ist falsch geschrieben. (Kompilierzeitfehler)

`Speicherkapazität reicht nicht aus`

Es wird mehr Speicher benötigt als verfügbar ist. Es kann zum Beispiel nicht genügend Speicher für die Zuweisung eines Dateipuffers vorhanden sein. Versuchen Sie, die Größe Ihrer DOS-Puffer zu verkleinern, entfernen Sie speicherresidente Programme, oder entfernen Sie einige Gerätetreiber. Wenn Sie große Datenfelder haben, versuchen Sie, zu Beginn Ihres Programms einen **$DYNAMIC**-Metabefehl einzusetzen. Wenn Sie Dokumente geladen haben, wird deren Entfernung einigen Speicher freigeben. (**bc**-Aufruf-, Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 7

`Steuerungsstruktur in IF...THEN...ELSE unvollständig`

Eine nicht passende **NEXT-**, **WEND-**, **END IF-**, **END SELECT-** oder **LOOP**-Anweisung erscheint in einer einzeiligen **IF...THEN...ELSE**-Anweisung. (Kompilierzeitfehler)

`STOP in Modul` *Name* `bei Adresse` *Segment:Offset*

Das Programm hat eine **STOP**-Anweisung erreicht. (Laufzeitfehler)

`SUB/FUNCTION ohne END SUB/FUNCTION`

Einer Prozedur fehlt die beendende Anweisung. (Kompilierzeitfehler)

`Syntaxfehler`

Diesen Fehler können verschiedene Bedingungen verursachen. Für die Kompilierzeit ist die häufigste Ursache ein falsch geschriebenes BASIC-Schlüsselwort oder -Argument. Während der Laufzeit wird er häufig durch eine unzulässig formatierte **DATA**-Anweisung verursacht. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code:2

`Syntaxfehler in numerischer Konstanten`

Eine numerische Konstante ist nicht richtig formatiert. Eine Beschreibung numerischer Konstanten finden Sie in Kapitel 2, "Datentypen". (Kompilierzeitfehler)

`Typ besteht aus mehr als 65535 Bytes`

Ein benutzerdefinierter Typ kann 64K nicht überschreiten. (Kompilierzeitfehler)

`Typ nicht definiert`

Das Argument *Benutzertyp* der **TYPE**-Anweisung ist nicht definiert. (Kompilierzeitfehler)

`TYPE ohne END TYPE`

Eine **TYPE**-Anweisung hat keine zugehörige **END TYPE**-Anweisung. (Kompilierzeitfehler)

`TYPE-Anweisung falsch verschachtelt`

Definitionen von benutzerdefinierten Typen sind in Prozeduren nicht zulässig. (Kompilierzeitfehler)

`Überlauf`

Das Ergebnis einer Berechnung ist zu groß, um innerhalb des für Gleitkommazahlen erlaubten Bereiches dargestellt zu werden. (Laufzeitfehler)

ERR-Code: 6

`Überlauf des Datenspeichers`

Die Programmdaten sind zu umfangreich und passen nicht in den Speicher. Dieser Fehler wird häufig durch zu viele Konstanten oder zu viele statische Datenfeld-Elemente verursacht. Wenn Sie den Befehl **bc** oder die Optionen **EXE-Datei erstellen** oder **Bibliothek erstellen** verwenden, versuchen Sie, Debug auszuschalten. Wenn der Speicher dann immer noch zu voll ist, teilen Sie Ihr Programm auf und verwenden Sie die **CHAIN**-Anweisung oder den **$DYNAMIC**-Metabefehl. (Kompilierzeitfehler)

`Überlauf des Kommunikationspuffers`

Während einer Datenfernübertragung ist der Empfangspuffer übergelaufen. Die Größe des Empfangspuffers wird durch die Befehlszeilenoption **/c** oder durch die Option **RB** in der **OPEN COM**-Anweisung gesetzt. Versuchen Sie, den Puffer häufiger zu prüfen (mit der **LOC**-Funktion) oder ihn häufiger zu leeren (mit der **INPUT$**-Funktion). (Laufzeitfehler)

ERR-Code: 69

`Überlauf des Programmspeichers`

Sie versuchen, ein Programm zu kompilieren, dessen Codesegment größer als 64K ist. Versuchen Sie, das Programm in einzelne Module aufzuteilen, oder verwenden Sie die **CHAIN**-Anweisung. (Kompilierzeitfehler)

`Überlauf in numerischer Konstanten`

Die numerische Konstante ist zu groß. (Kompilierzeitfehler)

`Überspringen bis END TYPE-Anweisung`

Ein Fehler in der **TYPE**-Anweisung hat QuickBASIC veranlaßt, alles zwischen der **TYPE**- und **END TYPE**-Anweisung zu ignorieren. (Kompilierzeitfehler)

`Umbenennen zwischen Disketten/Festplatte`

Es wurde versucht, eine Datei mit einer neuen Laufwerkskennung umzubenennen. Das ist nicht erlaubt. (Laufzeit-Anfrage)

ERR-Code: 74

`Unbekannte Anweisung`

Sie haben wahrscheinlich eine BASIC-Anweisung falsch geschrieben. (Kompilierzeitfehler)

`Unbekannte Option:` *Option*

Sie haben eine unzulässige Option angegeben. (**bc**-Aufruffehler)

`Unerwartetes Dateiende in TYPE-Deklaration`

Innerhalb eines **TYPE...END TYPE**-Blockes befindet sich ein Dateiendezeichen.

`Ungültige Konstante`

Ein ungültiger Ausdruck wird benutzt, um einer Konstanten einen Wert zuzuweisen. Erinnern Sie sich, daß Konstanten zugewiesene Ausdrücke, numerische Konstanten, symbolische Konstanten, jeden arithmetischen oder logischen Operator außer Exponentation enthalten dürfen. Ein Zeichenkettenausdruck, der einer Konstanten zugewiesen wird, darf nur aus einem einzigen Literal bestehen. (Kompilierzeitfehler)

`Ungültiges DECLARE für BASIC-Prozedur`

Sie versuchen, die Schlüsselworte **ALIAS, CDECL** oder **BYVAL** der **DECLARE**-Anweisung für eine BASIC-Prozedur zu benutzen. **ALIAS, CDECL** oder **BYVAL** können nur mit Nicht-BASIC-Prozeduren verwendet werden. (Kompilierzeitfehler)

`Ungültiges Zeichen`

QuickBASIC hat in der Quelldatei ein unzulässiges Zeichen, z. B. ein Steuerzeichen, gefunden. (Kompilierzeitfehler)

`Untere Grenze überschreitet obere Grenze`

Die untere Grenze überschreitet die in einer **DIM**-Anweisung definierte obere Grenze. (Kompilierzeitfehler)

`Unterprogramm nicht definiert`

Ein aufgerufenes Unterprogramm ist nicht definiert. (Kompilierzeitfehler)

`Unterprogramm unzulässig in Steueranweisungen`

FUNCTION-Definitionen sind in Unterprogrammen in steuernden Konstruktionen wie **IF...THEN...ELSE** und **SELECT CASE** unzulässig. (Kompilierzeitfehler)

`Unterprogrammfehler`

Dieser Fehler tritt meistens bei Unterprogramm- oder **FUNCTION**-Definitionen auf und kann eine der folgenden Ursachen haben:

- Das Unterprogramm oder **FUNCTION** ist bereits definiert.
- Das Programm enthält falsch geschachtelte **SUB**- oder **FUNCTION**-Anweisungen.
- Das Unterprogramm oder die **FUNCTION** wird nicht mit einer **END SUB**- oder **END FUNCTION**-Anweisung beendet.

(Kompilierzeitfehler)

`Unverträgliche Anzahl an Argumenten`

Sie verwenden eine falsche Anzahl von Argumenten mit einem BASIC-Unterprogramm oder einer BASIC-Funktion. (Kompilierzeitfehler)

`Unverträgliche Datentypen`

Die Variable ist nicht vom geforderten Typ. Sie versuchen beispielsweise, die **SWAP**-Anweisung mit einer Zeichenkettenvariablen und einer numerischen Variablen auszuführen. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 13

`Unverträgliche Parametertypen`

Der Typ eines Unterprogramm- oder **FUNCTION**-Parameters entspricht nicht dem **DECLARE**-Anweisungs-Argument oder dem aufrufenden Argument. (Kompilierzeitfehler)

`Unzulässig außerhalb eines TYPE-Blockes`

Die Klausel *Element* **AS** *Typ* ist nur innerhalb eines **TYPE...END TYPE**-Blockes erlaubt. (Kompilierzeitfehler)

`Unzulässig außerhalb von SUB, FUNCTION oder DEF FN`

Diese Anweisung ist im Modul-Ebenen-Code nicht erlaubt. (Kompilierzeitfehler)

`Unzulässig außerhalb von SUB/FUNCTION`

Die Anweisung ist im Modul-Ebenen-Code oder **DEF FN**-Funktionen nicht zulässig. (Kompilierzeitfehler)

`Unzulässig im Direktmodus`

Diese Anweisung ist nur innerhalb eines Programmes zulässig und kann nicht im Direkt-Fenster verwendet werden. (Kompilierzeitfehler)

`Unzulässige FOR-Indexvariable`

Dieser Fehler wird normalerweise verursacht, wenn ein falscher Variablentyp in einem **FOR**-Schleifenindex verwendet wird. Eine **FOR**-Schleifen-Indexvariable muß eine einfache numerische Variable sein. (Kompilierzeitfehler)

`Unzulässige Indexsyntax`

Ein Datenfeld-Index enthält einen Syntaxfehler: z. B. sowohl Zeichenketten- als auch ganzzahlige Datentypen. (Kompilierzeitfehler)

`Unzulässige Zahl`

Das Format der Zahl entspricht nicht dem im *BASIC-Befehlsverzeichnis* definierten Zahlenformat. Sie haben wahrscheinlich einen typographischen Fehler gemacht. Die Zahl 2p3 beispielsweise erzeugt diesen Fehler. (Kompilierzeitfehler)

`Unzulässiger COMMON-Name`

QuickBASIC hat eine unzulässige */Blockname/*-Angabe (z. B. ein *Blockname*, der ein BASIC-reserviertes Wort ist) in einem benannten **COMMON** entdeckt. (Kompilierzeitfehler)

`Unzulässiger Dateiname`

Bei **LOAD, SAVE, KILL** oder **OPEN** wird ein unzulässiges Format für den Dateinamen verwendet (z. B. ein Name mit zu vielen Zeichen). (Laufzeitfehler)

ERR-Code: 64

`Unzulässiger Funktionsaufruf`

Einer mathematischen oder Zeichenkettenfunktion wird ein Parameter übergeben, der außerhalb des Bereiches liegt. Außerdem kann ein Funktionsaufruffehler folgende Gründe haben:

- Es wird ein negativer oder übermäßig großer Index verwendet.
- Eine negative Zahl wird in eine Potenz erhoben, die keine ganze Zahl ist.
- Bei der Verwendung von **GET** *Datei* oder **PUT** *Datei* wird eine negative Datensatznummer angegeben.
- Ein ungültiges oder bereichsüberschreitendes Argument wird an eine Funktion übergeben.
- Eine **BLOAD**- oder **BSAVE**-Operation wird in ein Nicht-Diskettengerät geleitet.
- Eine E/A-Funktion oder -Anweisung (z. B. **LOC** oder **LOF**) wird für ein Gerät ausgeführt, das diese nicht unterstützt.
- Zeichenketten werden verkettet, und damit wird eine Zeichenkette gebildet, die länger als 32.767 Zeichen ist.

(Laufzeitfehler)

ERR-Code: 5

`Unzulässiger Funktionsname`

Ein in BASIC reserviertes Wort wird als Name einer vom Benutzer definierten **FUNCTION** verwendet. (Kompilierzeitfehler)

`Unzulässiges Trennzeichen`

In einer **PRINT USING**- oder **WRITE**-Anweisung steht ein unzulässiges Begrenzungszeichen. Verwenden Sie ein Semikolon oder ein Komma als Begrenzer. (Kompilierzeitfehler)

`Unzulässiges Typzeichen in einer numerischen Konstanten`

Eine numerische Konstante enthält ein unpassendes Typdeklarationszeichen. (Kompilierzeitfehler)

`Variable erforderlich`

QuickBASIC hat eine Anweisung **INPUT, LET, READ** oder **SHARED** ohne ein Variablenargument entdeckt. (Kompilierzeitfehler)

`Variable erforderlich`

Eine **GET**- oder **PUT**-Anweisung hat bei der Bearbeitung einer im BINARY-Modus geöffneten Datei keine Variable angegeben. (Laufzeitfehler)

ERR-Code: 40

`Variablenname bereits vorhanden`

Sie versuchen, *x* als benutzerdefinierten Typ zu definieren, nachdem bereits *x.y* verwendet wird. (Kompilierzeitfehler)

`Verbund/Zeichenketten-Zuweisung erforderlich`

Der Anweisung **LSET** fehlt die Zuweisung der Zeichenketten- oder Verbundvariablen. (Kompilierzeitfehler)

`Verschachtelte Funktionsdefinition`

Eine **FUNCTION**-Definition erscheint in einer anderen **FUNCTION**-Definition oder innerhalb einer **IF...THEN...ELSE**-Klausel. (Kompilierzeitfehler)

`Wählen Sie Neu aus dem Menü Bearbeiten für neue SUB/FUNCTION`

Sie versuchen, eine Prozedur in einem Fenster auf Modul-Ebene einzugeben. In QuickBASIC hat jedes Unterprogramm und jede **FUNCTION**-Prozedur ihren eigenen Arbeitsbereich. (Kompilierzeitfehler)

`WEND ohne WHILE`

Dieser Fehler wird verursacht, wenn eine **WEND**-Anweisung keine entsprechende **WHILE**-Anweisung hat. (Kompilierzeitfehler)

`WHILE ohne WEND`

Dieser Fehler wird verursacht, wenn eine **WHILE**-Anweisung keine entsprechende **WEND**-Anweisung hat. (Kompilierzeitfehler)

`Zeichenkette variabler Länge erforderlich`

In einer **FIELD**-Anweisung sind nur Zeichenketten variabler Länge erlaubt. (Kompilierzeitfehler)

`Zeichenketten fester Länge unzulässig`

Sie versuchen, eine Zeichenkette fester Länge als formalen Parameter zu benutzen. (Kompilierzeitfehler)

`Zeichenkettenausdruck erforderlich`

Die Anweisung erfordert als Argument einen Zeichenkettenausdruck. (Kompilierzeitfehler)

`Zeichenkettenbereich beschädigt`

Dieser Fehler tritt auf, wenn während einer Heap-Komprimierung aus dem Zeichenkettenbereich eine ungültige Zeichenkette gelöscht wird. Wahrscheinliche Ursachen für diesen Fehler sind die folgenden:

- Ein Zeichenkettenbeschreiber oder Zeichenkettenrückzeiger ist unzulässig verändert worden. Dies kann eintreten, wenn Sie eine Assembler-Unterroutine verwenden, um Zeichenketten zu modifizieren.
- Außerhalb des Bereichs liegende Datenfeldindizes werden verwendet, und der Zeichenkettenbereich wird unabsichtlich verändert. Die Option "Erzeuge Debug-Code" kann während der Kompilierzeit verwendet werden, um nach Datenfeldindizes zu suchen, die die Datenfeldgrenzen überschreiten.
- Der unkorrekte Gebrauch der Anweisungen **POKE** und/oder **DEF SEG** kann den Zeichenkettenbereich unzulässig verändern.
- Zwischen zwei verketteten Programmen können nicht-übereinstimmende **COMMON**-Deklarationen auftreten.

(Laufzeitfehler)

`Zeichenkettenformel zu umfangreich`

Eine Zeichenkettenformel ist zu lang, oder eine **INPUT**-Anweisung erfordert mehr als 15 Zeichenkettenvariablen. Teilen Sie die Formel oder die **INPUT**-Anweisung zur korrekten Ausführung auf. (Laufzeitfehler)

ERR-Code:16

`Zeichenkettenvariable erforderlich`

Die Anweisung erfordert als Argument eine Zeichenkettenvariable. (Kompilierzeitfehler)

`Zeile ungültig. Erneut starten.`

Es wird ein ungültiges Dateinamen-Zeichen nach den Pfadzeichen "\" (umgekehrter Schrägstrich) oder ":" (Doppelpunkt) benutzt. (**bc**-Aufruffehler)

`Zeile zu lang`

Zeilenlängen sind auf 255 Zeichen begrenzt. (Kompilierzeitfehler)

`Zu viele Argumente im Funktionsaufruf`

Funktionsaufrufe sind auf 60 Argumente begrenzt. (Kompilierzeitfehler)

`Zu viele benannte COMMON-Blöcke`

Die erlaubte maximale Anzahl benannter **COMMON**-Blöcke beträgt 126. (Kompilierzeitfehler)

`Zu viele Dateien`

Während der Kompilierzeit tritt dieser Fehler auf, wenn Include-Dateien über mehr als fünf Ebenen verschachtelt sind. Er tritt während der Laufzeit auf, wenn das Verzeichnis-Maximum von 255 Dateien mit dem Versuch, eine neue Datei mit einer **SAVE**- oder **OPEN**-Anweisung anzulegen, überschritten wird. (Kompilierzeit- oder Laufzeitfehler)

ERR-Code: 67

`Zu viele Dimensionen`

Datenfelder sind auf 60 Dimensionen begrenzt. (Kompilierzeitfehler)

`Zu viele Marken`

Die Zeilenanzahl in der Zeilenliste, die der **ON...GOTO**- oder **ON...GOSUB**-Anweisung folgt, überschreitet 255. (Kompilierzeitfehler)

`Zu viele TYPE-Definitionen`

Die maximal erlaubte Anzahl von benutzerdefinierten Typen beträgt 240. (Kompilierzeitfehler)

`Zu viele Variablen für INPUT`

Eine **INPUT**-Anweisung ist auf 60 Variablen begrenzt. (Kompilierzeitfehler)

`Zu viele Variablen für LINE INPUT`

In einer **LINE INPUT**-Anweisung ist nur eine Variable erlaubt. (Kompilierzeitfehler)

`Zugriff nicht gestattet`

Es wurde versucht, auf eine schreibgeschützte Diskette zu schreiben oder auf eine gesperrte Datei zuzugreifen. (Laufzeitfehler)

ERR-Code:70

`Zusätzlicher Dateiname nicht beachtet`

In der Befehlszeile sind zu viele Dateinamen angegeben, der letzte Dateiname wurde nicht beachtet. (**bc**-Aufruffehler)

`Zuweisung einer Zeichenkette erforderlich`

Bei einer **RSET**-Anweisung fehlt die Zuweisung der Zeichenkette. (Kompilierzeitfehler)

D.2 Linker-Fehlermeldungen

Dieser Abschnitt enthält eine Auflistung und Beschreibung über die vom Microsoft Overlay-Linker **LINK** erzeugten Fehlermeldungen.

Fatale Fehler veranlassen den Linker, die Ausführung abzubrechen. Meldungen zu fatalen Fehlern haben die folgende Form:

Dateilage : `Fehler L1`*xxx*: *Text der Meldung*

Nichtfatale Fehler zeigen Probleme in der ausführbaren Datei an. **LINK** erstellt die ausführbare Datei. Meldungen zu nichtfatalen Fehlern haben die folgende Form:

Dateilage : `Fehler L2`*xxx*: *Text der Meldung*

Warnungen zeigen mögliche Probleme in ausführbaren Dateien an. **LINK** erstellt die ausführbare Datei. Warnungen haben die folgende Form:

Dateilage : `Warnung L4`*xxx*: *Text der Meldung*

In diesen Meldungen ist *Dateilage* die mit dem Fehler verknüpfte Eingabedatei oder **LINK**, wenn keine Eingabedatei vorhanden ist.

Die folgenden Fehlermeldungen können erscheinen, wenn Sie Objektdateien mit dem Microsoft Overlay-Linker **LINK** binden:

Nummer	*Linker-Fehlermeldung*
L1001	*Option* : `Optionsname nicht eindeutig` Es erscheint kein eindeutiger Optionsname nach dem Optionsanzeiger (/). Dieser Fehler wird z.B. durch den Befehl `LINK /N main;` erzeugt, da **LINK** nicht erkennen kann, welche der drei Optionen, die mit "N" beginnen, gewünscht war.
L1002	*Option* : `Unbekannter Optionsname` Ein unbekanntes Zeichen folgt dem Optionsanzeiger (/), wie im folgenden Beispiel: `LINK /ABCDEF main;`
L1003	`/QUICKLIB, /EXEPACK nicht kompatibel` Sie haben beide Optionen /QUICKLIB und /EXEPACK angegeben; diese beiden Optionen können aber nicht zusammen verwendet werden.
L1004	*Option* : `Ungültiger numerischer Wert` Für eine der Linker-Optionen erscheint ein falscher Wert. Zum Beispiel ist eine Zeichenkette für eine Option angegeben, die jedoch einen numerischen Wert erfordert.
L1006	*Option* : `Stapelgröße überschreitet 65535 Bytes` Der Wert, der als Parameter der Option /STACKSIZE übergeben wurde, überschreitet den maximal möglichen Wert.
L1007	*Option* : `Interrupt-Nummer überschreitet 255` Eine Zahl größer als 255 wurde als Wert für die Option **/OVERLAYINTERRUPT** angegeben.
L1008	*Option* : `Segmentlimit zu groß` Beim Verwenden der Option **/SEGMENTS** ist die erlaubte Höchstzahl an Segmenten größer als 3072 gesetzt.

L1009 *Nummer*: `CPARMAXALLOC: Unzulässsiger Wert`

Die Zahl, die in der Option **/CPARMAXALLOC** angegeben wird, befindet sich nicht im Bereich von 1 - 65535.

L1020 `Objektmodule nicht festgelegt`

Dem Linker wurden keine Objektdateinamen angegeben.

L1021 `Antwortdateien können nicht verschachtelt werden`

Eine Antwortdatei (Response File) erscheint innerhalb einer Antwortdatei.

L1022 `Antwortzeile zu lang`

Eine Zeile in einer Antwortdatei ist länger als 127 Zeichen.

L1023 `Vom Benutzer abgebrochen`

Sie haben STRG+C eingegeben.

L1024 `Verschachtelte rechte Klammern`

Der Inhalt eines Overlays wurde auf der Befehlszeile falsch eingegeben.

L1025 `Verschachtelte linke Klammern`

Der Inhalt eines Overlays wurde auf der Befehlszeile falsch eingegeben.

L1026 `Fehlende rechte Klammer`

Eine rechte Klammer fehlt in der Inhaltsbeschreibung eines Overlays auf der Befehlszeile.

L1027 `Fehlende linke Klammer`

Eine linke Klammer fehlt in der Inhaltsbeschreibung eines Overlays auf der Befehlszeile.

L1043 `Überlauf der Verschiebungstabelle`

Es erscheinen mehr als 32768 lange Aufrufe, lange Sprünge oder andere lange Zeiger in dem Programm.

Versuchen Sie, lange Verweise nach Möglichkeit durch kurze Verweise zu ersetzen, und erstellen Sie das Objektmodul erneut.

L1045 `Zu viele TYPDEF-Sätze`

Ein Objektmodul enthält mehr als 255 **TYPDEF**-Sätze. Diese Sätze beschreiben gemeinsame Variablen. Dieser Fehler kann nur bei Programmen auftreten, die mit dem Microsoft FORTRAN-Compiler oder anderen Compilern erstellt wurden, die gemeinsame Variablen unterstützen. (**TYPDEF** ist ein DOS-Ausdruck. Er wird im *Microsoft MS-DOS Programmer's Reference* und anderen DOS-Nachschlage-werken erklärt.)

L1046 `Zu viele externe Symbole in einem Modul`

Ein Objektmodul legt die Höchstgrenze der externen Symbole auf über 1023 fest.

Teilen Sie das Modul in kleinere Abschnitte auf.

L1047 `Zu viele Gruppen-, Segment- und Klassennamen in einem Modul`

Das Programm enthält zuviele Gruppen-, Segment- und Klassennamen.

Verringern Sie die Anzahl der Gruppen, Segmente und Klassen, und erstellen Sie die Objektdatei erneut.

L1048 `Zu viele Segmente in einem Modul`

Eine Objektmodul hat mehr als 255 Segmente.

Teilen Sie das Modul oder fügen Sie Segmente zusammen.

L1049 `Zu viele Segmente`

Das Programm enthält mehr als die maximale Anzahl von Segmenten. (Die Option **/SEGMENTS** legt die maximal zulässige Anzahl fest; der Standardwert ist 128.)

Binden Sie erneut, indem Sie die Option **/SEGMENTS** mit einer passenden Anzahl von Segmenten verwenden.

L1050 `Zu viele Gruppen in einem Modul`

LINK stellte mehr als 21 Gruppen-Definitionen (**GRPDEF**) in einem einzigen Modul fest.

Verringern Sie die Anzahl der Gruppen-Definitionen oder teilen Sie das Modul. (Gruppen-Definitionen werden in dem *Microsoft MS-DOS Programmer's Reference* und in anderen Nachschlagewerken zu DOS erklärt.)

L1051 `Zu viele Gruppen`

Das Programm definiert mehr als 20 Gruppen, **DGROUP** nicht mitgezählt.

Verringern Sie die Anzahl der Gruppen.

L1052 `Zu viele Bibliotheken`

Es wurde der Versuch gemacht, mit mehr als 32 Bibliotheken zu binden.

Kombinieren Sie Bibliotheken, oder verwenden Sie Module, die weniger Bibliotheken erfordern.

L1053 `Symboltabellen-Überlauf`

Es gibt keine feste Obergrenze für die Größe der Symboltabelle. Diese ist jedoch durch die Größe des verfügbaren Speicherplatzes beschränkt.

Kombinieren Sie Module oder Segmente und erstellen Sie die Objektdatei erneut. Entfernen Sie so viele globale Symbole wie möglich.

L1054 `Erfordertes Segmentlimit zu hoch`

Der Linker hat nicht genügend Speicher, um Tabellen zuzuweisen, die die Anzahl der erforderten Segmente beschreiben. (Die Standard-einstellung ist 128 oder der Wert, der mit der Option **/SEGMENTS** festgelegt wird.)

Versuchen Sie erneut zu binden, indem Sie die Option **/SEGMENTS** verwenden, um eine kleinere Anzahl von Segmenten auszuwählen (z. B. benutzen Sie 64, wenn Sie zunächst die Standardeinstellung benutzt haben) oder schaffen Sie Speicherplatz, indem Sie speicherresidente Programme oder Shells entfernen.

L1056 `Zu viele Overlays`

Das Programm definierte mehr als 63 Overlays.

L1057 `Datensatz zu lang`

Ein **LEDATA**-Satz (in einem Objektmodul) enthält mehr als 1024 Byte Daten. Dies ist ein Übersetzerfehler. (**LEDATA** ist ein DOS-Ausdruck, der im *Microsoft MS-DOS Programmer's Reference* und in anderen DOS-Nachschlagewerken erklärt wird.)

L1070 `Segmentgröße überschreitet 64K`

Ein einzelnes Segment enthält mehr als 64K Code oder Daten. Versuchen Sie, unter Verwendung des Large-Modells zu kompilieren und zu binden.

L1071 `Segment _TEXT größer als 65520 Bytes`

Dieser Fehler tritt wahrscheinlich nur in C-Programmen vom Typ Small-Modellauf. Er kann jedoch auch auftreten, wenn Programme mit einem Segment namens **_TEXT**, unter Verwendung der Option **/DOSSEG**, gebunden werden. C-Programme vom Typ Small-Modell müssen die Code-Adressen 0 und 1 reservieren; dieser Bereich wird für Ausrichtungszwecke auf 16 erhöht.

L1072 `Common-Bereich größer als 65536 Bytes`

Das Programm enthält mehr als 64K gemeinsamer Variablen. Dieser Fehler kann nicht in Verbindung mit Objektdateien auftreten, die mit dem Microsoft Macro-Assembler (**MASM**) erstellt wurden. Dieser Fehler tritt nur bei Programmen auf, die von Compilern erstellt wurden, die gemeinsame Variablen unterstützen.

L1080 `List-Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll. Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1081 `Kein Speicherplatz für die zu startende Datei vorhanden`

Die Diskette/Festplatte, auf die die **.EXE**-Datei geschrieben werden soll, ist voll.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte und starten Sie den Linker erneut.

L1083 `Zu startende Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1084 `Temporäre Datei kann nicht erstellt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Schaffen Sie Platz in dem Verzeichnis und starten Sie den Linker erneut.

L1085 `Temporäre Datei kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

L1086 `Temporäre Datei fehlt`

Ein interner Fehler ist aufgetreten.

L1087 `Unerwartetes Dateiende auf temporärer Datei`

Die Diskette mit der temporären Linker-Ausgabedatei wurde entfernt.

L1088 `Kein Speicherplatz für List-Datei vorhanden`

Die Diskette/Festplatte, auf die die Listdatei geschrieben wird, ist voll.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte und starten Sie den Linker erneut.

L1089 *Dateiname* `: Antwortdatei kann nicht geöffnet werden`

LINK konnte die angegebene Antwortdatei nicht finden.

Dies zeigt normalerweise einen Schreibfehler an.

L1090 `List-Datei kann nicht erneut geöffnet werden`

Die ursprüngliche Diskette wurde bei der Anfrage nicht wieder eingelegt.

Starten Sie den Linker erneut.

L1091 `Unerwartetes Dateiende in Bibliothek`

Die Diskette, die die Bibliothek enthält, ist wahrscheinlich entfernt worden.

Legen Sie die Diskette wieder ein, und starten Sie den Linker erneut.

L1093 `Objekt nicht gefunden`

Eine der in der Linker-Eingabe angegebenen Objektdateien wurde nicht gefunden.

Starten Sie den Linker erneut und geben Sie die Objektdatei an.

L1101 `Ungültiges Objektmodul`

Eines der Objektmodule ist ungültig.

L1102 `Unerwartetes Dateiende`

Es tritt ein ungültiges Format für eine Bibliothek auf.

L1103 `Versuch, auf Daten außerhalb der Segmentgrenzen zuzugreifen`

Ein Datensatz in einem Objektmodul gibt Daten an, die über das Segmentende hinausgehen. Dies ist ein Übersetzerfehler.

L1104 *Dateiname* `: Ungültige Bibliothek`

Die angegebene Datei ist keine gültige Bibliotheksdatei. Dieser Fehler veranlaßt **LINK** abzubrechen.

L1113 `Nicht aufgelöste COMDEF; interner Fehler`

L1114 `Datei nicht passend zu /EXEPACK; binden Sie erneut ohne`

Für das gebundene Programm ist das gepackte Ladeabbild plus Pack-Verwaltung größer als das ungepackte Ladeabbild.

Binden Sie erneut ohne die Option **/EXEPACK.**

L1115 `/QUICKLIB, Overlays nicht kompatibel`

Sie haben Overlays spezifiziert und verwenden die Option /QUICKLIB. Sie können jedoch nicht beide verwenden.

L1123 *Segmentname* `: Segment definiert 16 und 32 Bit`

Siehe Dokumentation zu Microsoft Macro Assembler 5.0.

L2001 `Fixup(s) ohne Daten`

Ein **FIXUPP**-Satz kommt ohne einen ihm direkt vorhergehenden Datensatz vor. Dies ist wahrscheinlich ein Compilerfehler. (Im *Microsoft MS-DOS Programmer's Reference* finden Sie weitere Informationen zu **FIXUPP**.)

`L2002` `Fixup-Überlauf nahe` *Zahl* `im Segment` *Segname*

Die folgenden Bedingungen können diesen Fehler verursachen:

- Eine Gruppe ist größer als 64K.
- Das Programm enthält einen kurzen Intersegment-Sprung oder kurzen Intersegment-Aufruf.
- Der Name eines Datenobjekts in dem Programm kollidiert mit dem einer Bibliotheksunterroutine, die in das gebundene Programm einbezogen ist.
- Eine **EXTRN**-Deklaration in einer Assembler-Quelldatei erscheint innerhalb eines Segmentkörpers, wie im folgenden Beispiel:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Die folgende Konstruktion ist vorzuziehen:

```
        EXTRN   main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Überprüfen Sie die Quelldatei und erstellen Sie die Objektdatei erneut. (Weitere Informationen zu Rahmen- und Ziel-Segmenten finden Sie im *Microsoft MS-DOS Programmmer's Reference.)*

`L2005` `Fixup-Typ nicht unterstützt nahe` *Zahl* `im Segment` *Segmentname*

Ein Fixup-Typ tritt auf, der nicht von dem Microsoft-Linker unterstützt wird. Dies ist wahrscheinlich ein Compilerfehler.

L2011 *Name* : `NEAR/HUGE-Konflikt`

Einer gemeinsamen Variablen sind kollidierende **NEAR**- und **HUGE**-Attribute gegeben worden. Dieser Fehler kann nur bei Programmen auftreten, die mit Compilern erstellt wurden, die gemeinsame Variablen unterstützen.

L2012 *Name* : `Unverträgliche Datenfeldelementgröße`

Ein langes gemeinsames Datenfeld wird mit zwei oder mehr verschiedenen Datenfeldelementgrößen deklariert. (Zum Beispiel wurde ein Datenfeld einmal als Datenfeld mit Zeichen und einmal als Datenfeld mit reellen Zahlen deklariert.) Dieser Fehler kann nicht bei Objektdateien auftreten, die mit dem Microsoft-Macro-Assembler erstellt wurden. Er tritt nur bei Compilern auf, die lange gemeinsame Datenfelder unterstützen.

L2013 `LIDATA-Satz zu groß`

Ein **LIDATA**-Satz enthält mehr als 512 Bytes. Dieser Fehler wird normalerweise durch einen Compiler-Fehler verursacht.

L2024 *Name* : `Symbol bereits definiert`

Der Linker hat ein offensichtlich globales Symbol gefunden, welches erneut definiert wurde. Entfernen Sie die andere(n) Definition(en).

L2025 *Name* : `Symbol mehr als einmal definiert`

Entfernen Sie die doppelte Symboldefinition aus der Objektdatei.

L2029 `Nicht aufgelöstes externes Symbol`

Ein oder mehrere Symbole sind in einem oder mehreren Modulen als extern deklariert, sind aber in keinem der Module oder keiner der Bibliotheken als global definiert. Eine Liste der nicht aufgelösten externen Verweise erscheint, wie im nachfolgenden Beispiel gezeigt, nach der Meldung.

```
 Nicht aufgelöste externe Symbole
EXIT in Datei(en):
 HAUPT.OBJ (haupt.for)
OPEN in Datei(en):
 HAUPT.OBJ (haupt.for)
```

Der Name, der vor `in Datei(en)` steht, ist das nicht aufgelöste externe Symbol. Die nächste Zeile enthält eine Liste der Objektmodule, die Bezug auf dieses Symbol nehmen. Diese Meldung und die Liste werden ebenfalls in die Map-Datei geschrieben, sofern eine vorhanden ist.

L2041 `Stapel plus Daten überschreiten 64K`

Die Gesamtgröße von Stapel und Near-Daten überschreiten 64K. Reduzieren Sie die Größe des Stapels, um diesen Fehler zu beheben.

Der Linker testet diese Bedingung nur, wenn die Option **/DOSSEG** gesetzt wurde. Diese Option wird automatisch durch das Bibliotheks-startmodul gesetzt.

L2043 `Quick-Bibliotheks-Einrichtungsmodul fehlt`

Sie haben nicht angegeben, bzw. **LINK** kann das Objektmodul oder die Bibliothek zur Einrichtung einer Quick-Bibliothek nicht finden. Im Fall von QuickBASIC wird die Bibliothek durch **BQLB40.LIB** bereitgestellt.

L2044 *Name* `: Symbol bereits definiert; /NOE verwenden`

Der Linker hat eine erneute Definition eines offensichtlich globalen Symbols gefunden. Dieser Fehler wird oft dadurch verursacht, daß ein in der Bibliothek definiertes Symbol wieder definiert wird. Binden Sie erneut unter Verwendung der Option **/NOEXTDICTIONARY /NOE**).

Dieser Fehler, zusammen mit dem Fehler `L2025` für dasselbe Symbol, zeigen einen realen Redefinitions-Fehler an.

L4012 `HIGH Laden schaltet EXEPACK aus`

Die Optionen **/HIGH** und **/EXEPACK** können nicht gleichzeitig verwendet werden.

L4015 `/CODEVIEW schaltet /DSALLOCATE aus`

Die Optionen **/CODEVIEW** und **/DSALLOCATE** können nicht gleichzeitig verwendet werden.

L4016 `/CODEVIEW schaltet /EXEPACK aus`

Die Optionen **/CODEVIEW** und **/EXEPACK** können nicht gleichzeitig verwendet werden.

L4020 *Name* : `Codesegmentgröße überschreitet 65500`

Codesegmente von 65501 bis 65536 Bytes Länge können beim Intel 80286-Prozessor unzuverlässig sein.

L4021 `Kein Stapelsegment`

Das Programm enthält kein Stapelsegment, das mit dem Kombiniertyp **STACK** definiert ist. Diese Meldung dürfte für Module, die mit Microsoft QuickBASIC kompiliert sind, nicht erscheinen, kann aber bei einem Assembler-Modul auftreten.

Normalerweise sollte jedes Programm ein Stapelsegment mit dem Kombiniertyp **STACK** haben. Sie können diese Meldung ignorieren, wenn Sie einen besonderen Grund haben, keinen Stapel bzw. einen Stapel ohne den Kombinier-Typ **STACK** zu definieren. Das Binden mit Linker-Versionen kleiner als Version 2.40 kann diese Meldung verursachen, da diese Linker die Bibliotheken nur einmal durchsuchen.

L4031 *Name* : `Segment in mehr als einer Gruppe deklariert`

Ein Segment ist als Teil zweier verschiedener Gruppen deklariert.

Korrigieren Sie die Quelldatei, und erstellen Sie die Objektdateien erneut.

L4034 `Mehr als 239 Overlay-Segmente; Extras im Stamm`

Das Programm setzt mehr als 239 Segmente in die Overlays. Wenn dieser Fehler auftritt, werden die Segmente beginnend mit der Nummer 234 im permanent residenten Teil, dem Stamm, plaziert.

L4045 `Name der Ausgabedatei ist` *Name* :

Die Anfrage (Prompt) für die ausführbare Datei gibt einen fehlerhaften Standardwert an, weil **/QUICKLIB** nicht früh genug verwendet wurde. Die Ausgabe ist eine Quick-Bibliothek mit dem in der Fehlermeldung angegebenen Namen.

L4050 `Zu viele globale Symbole zu sortieren`

Die Anzahl der globalen Symbole überschreitet den für die mit der Option **/MAP** angeforderten Speicherplatz für die Sortierung der Symbole. Die Symbole bleiben unsortiert.

L4051	*Dateiname* `: Kann Bibliothek nicht finden`
	Der Linker kann die angegebene Datei nicht finden.
	Geben Sie entweder einen neuen Dateinamen, eine neue Pfadbeschreibung oder beides ein.
L4053	`VM.TMP : unzulässiger Dateiname; ignoriert`
	VM.TMP erscheint als Name einer Objektdatei. Benennen Sie die Datei um, und starten Sie den Linker erneut.
L4054	*Dateiname* `: Kann Datei nicht finden`
	Der Linker kann die angegebene Datei nicht finden. Geben Sie entweder einen neuen Dateinamen, eine neue Pfadbeschreibung oder beides ein.

D.3 LIB-Fehlermeldungen

Fehlermeldungen, die vom Microsoft-Bibliotheksmanager **LIB** erzeugt werden, haben eines der folgenden Formate:

{*Dateiname* | `LIB`} : `fataler Fehler U1`*xxx* : *Text der Meldung*

{*Dateiname* | `LIB`} : `Fehler U2`*xxx* : *Text der Meldung*

{*Dateiname* | `LIB`} : `Warnung U4`*xxx* : *Text der Meldung*

Die Fehlermeldung beginnt mit dem Namen der Eingabedatei (*Dateiname*), sofern eine existiert, oder mit dem Namen des Dienstprogrammes. Wenn möglich, schreibt **LIB** eine Warnung und fährt mit der Ausführung fort. In einigen Fällen sind Fehler fatal und **LIB** beendet die Verarbeitung. **LIB** kann die folgenden Fehlermeldungen anzeigen:

Nummer	***LIB-Fehlermeldung***
U1150	`Seitengröße zu klein`
	Die Seite einer eingegebenen Bibliothek war zu klein. Dies zeigt eine ungültig eingegebene **.LIB**-Datei an.
U1151	`Syntaxfehler : Ungültige Dateiangabe`
	Ein Befehlsoperator, wie z. B. ein Minuszeichen (-), wird ohne einen darauf folgenden Modulnamen angegeben.
U1152	`Syntaxfehler : Fehlender Optionsname`
	Ein Schrägstrich (/) wird ohne eine ihm folgende Option angegeben.

U1153 `Syntaxfehler : Fehlender Optionswert`

Die Option **/PAGESIZE** wird ohne einen ihr folgenden Wert angegeben.

U1154 `Option unbekannt`

Es wird eine unbekannte Option eingegeben. Üblicherweise erkennt **LIB** nur die Option **/PAGESIZE**.

U1155 `Syntaxfehler : Unzulässige Eingabe`

Der eingegebene Befehl entspricht nicht der korrekten **LIB**-Syntax, wie sie in dem *Microsoft CodeView™ and Utilities manual*, Kapitel 13, "Managing Libraries with LIB", angegeben ist.

U1156 `Syntaxfehler`

Der eingegebene Befehl entspricht nicht der korrekten **LIB**-Syntax, wie sie in dem *Microsoft CodeView™ and Utilities manual*, Kapitel 13, "Managing Libraries with LIB ", angegeben ist.

U1157 `Komma oder neue Zeile fehlt`

Ein Komma oder ein Wagenrücklauf wird in der Befehlszeile erwartet, erscheint jedoch nicht. Dies kann auf ein falsch plaziertes Komma hinweisen, wie in der folgenden Zeile:

```
LIB math.lib,-mod1+mod2;
```

Die Zeile hätte wie folgt eingegeben werden müssen:

```
LIB math.lib -mod1+mod2;
```

U1158 `Fehlendes Endezeichen`

Entweder die Antwort auf die Anfrage `Ausgabe-Bibliothek` oder die letzte Zeile der Antwortdatei, die zum Start von **LIB** benutzt wird, wird nicht mit der EINGABETASTE abgeschlossen.

U1161 `Alte Bibliothek kann nicht umbenannt werden`

LIB kann die alte Bibliothek mit der Erweiterung **.BAK** nicht umbenennen, da die **.BAK**-Version bereits mit einem Schreibschutz versehen existiert.

Ändern Sie den Schutz der alten **.BAK**-Version.

U1162 `Bibliothek kann nicht erneut geöffnet werden`

Die alte Bibliothek kann nicht erneut geöffnet werden, nachdem sie mit einer **.BAK**-Erweiterung umbenannt ist.

U1163 `Fehler beim Schreiben in Datei mit externen Querverweisen`

Diskette/Festplatte oder Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1170 `Zu viele Symbole`

Es erscheinen mehr als 4609 Symbole in der Bibliotheksdatei.

U1171 `Unzureichender Speicherplatz`

LIB hat nicht genug Speicher, um zu starten.

Beseitigen Sie alle Shells oder speicherresidente Programme, und versuchen Sie abermals zu starten, oder fügen Sie mehr Speicherplatz hinzu.

U1172 `Kein virtueller Speicher mehr vorhanden`

U1173 `Interner Fehler`

U1174 `Kennzeichen : Nicht zugewiesen`

U1175 `Frei : Nicht zugewiesen`

U1180 `Schreiben in Auszugsdatei nicht möglich`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1181 `Schreiben in Bibliotheksdatei nicht möglich`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1182 — *Dateiname* : `Auszugsdatei kann nicht angelegt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll oder die angegebene Auszugsdatei existiert bereits mit einem Schreibschutz.

Schaffen Sie Speicherplatz auf der Diskette/Festplatte oder verändern Sie den Schutz der Auszugsdatei.

U1183 — `Antwortdatei kann nicht geöffnet werden`

Die Antwortdatei wurde nicht gefunden.

U1184 — `Unerwartetes Dateiende bei Befehlseingabe`

Ein Dateiendezeichen wurde zu früh als Antwort auf eine Anfrage empfangen.

U1185 — `Neue Bibliothek kann nicht erstellt werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll oder die Bibliotheksdatei existiert bereits mit einem Schreibschutz.

Schaffen Sie Platz auf der Diskette/Festplatte oder in dem Verzeichnis oder entfernen Sie den Schreibschutz.

U1186 — `Fehler beim Schreiben in neue Bibliothek`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1187 — `VM.TMP kann nicht geöffnet werden`

Die Diskette/Festplatte oder das Hauptverzeichnis ist voll.

Löschen oder entfernen Sie Dateien, um Speicherplatz zu schaffen.

U1188 — `In VM kann nicht geschrieben werden`

U1189 — `Aus VM kann nicht gelesen werden`

U1200 — *Name* : `Ungültiger Bibliothekskopf`

Die eingegebene Bibliotheksdatei hat ein ungültiges Format. Sie ist entweder keine Bibliotheksdatei, oder sie ist fehlerhaft.

U1203 — *Name* : `Ungültiges Objektmodul nahe` *Dateilage*

Das mit *Name* gekennzeichnete Modul ist kein gültiges Objektmodul.

U2152 — *Dateiname* : `Listing kann nicht erstellt werden`

Das Verzeichnis oder die Diskette/Festplatte ist voll, oder die Listingdatei der Querverweise existiert bereits mit Schreibschutz. Schaffen Sie Platz auf der Diskette/Festplatte oder ändern Sie den Schutz der Listingdatei der Querverweise.

U2155 — *Modulname* : `Modul nicht in Bibliothek; ignoriert`

Das angegebene Modul wurde nicht in der eingegebenen Bibliothek gefunden.

U2157 — *Dateiname* : `Auf Datei kann nicht zugegriffen werden`

LIB kann die angegebene Datei nicht öffnen.

U2158 — *Bibliotheksname* : `Ungültiger Bibliothekskopf; Datei ignoriert`

Die eingegebene Bibliothek hat ein ungültiges Format.

U2159 — *Dateiname* : `Ungültiges Format` *Hexzahl*; `Datei ignoriert`

Das Signatur-Byte oder -Wort *Hexzahl* der gegebenen Datei ist nicht einer der folgenden anerkannten Typen: Microsoft-Bibliothek, Intel-Bibliothek, Microsoft-Objekt, Xenix-Archiv.

U4150 — *Modulname* : `Erneute Moduldefinition wird ignoriert`

Ein Modul wurde festgelegt, um in eine Bibliothek eingefügt zu werden, aber ein Modul desselben Namens existiert bereits in der Bibliothek. Oder wenn ein Modul mit dem gleichen Namen mehr als einmal in der Bibliothek gefunden wurde.

U4151 — *Symbol*: `Symbol im Modul` *Modulname* `erneut definitiert; Redefinition ignoriert`

Das angegebene Symbol wird in mehr als einem Modul definiert.

U4153 — *Zahl* : `Seitengröße zu klein; ignoriert`

Der mit der Option **/PAGESIZE** angegebene Wert ist kleiner als 16.

U4155 *Modulname* : `Modul ist nicht in der Bibliothek`

Ein zu ersetzendes Modul ist nicht in der Bibliothek. **LIB** fügt das Modul der Bibliothek hinzu.

U4156 *Bibliotheksname* : `Angabe der Ausgabebibliothek ignoriert`

Eine Ausgabebibliothek ist zusätzlich zu einem neuen Bibliotheksnamen angegeben. Wenn zum Beispiel

```
LIB new.lib+one.obj,new.lst,new.lib
```

angegeben wird, während `new.lib` noch nicht existiert, tritt dieser Fehler auf.

U4157 `Unzureichender Speicherplatz; erweitertes Verzeichnis nicht erzeugt`

Diese LIB-Warnung zeigt an, daß LIB kein erweitertes Verzeichnis erzeugen konnte. Die Bibliothek ist weiterhin gültig, aber der Linker kann den Vorteil des erweiterten Verzeichnisses zum schnelleren binden nicht nutzen.

U4158 `Interner Fehler; erweitertes Verzeichnis nicht erzeugt`

Diese LIB-Warnung zeigt an, daß LIB kein erweitertes Verzeichnis erzeugen konnte. Die Bibliothek ist weiterhin gültig, aber der Linker kann den Vorteil des erweiterten Verzeichnisses zum schnelleren binden nicht nutzen.

Index

E

Dokumentationsbeurteilung

Wir legen großen Wert auf Ihre Meinung, weil wir in Zukunft unsere Produkte entsprechend Ihren Vorstellungen herausbringen wollen. Bitte teilen Sie uns mit, was Sie von dieser Dokumentation halten, indem Sie diese Karte ausfüllen und an uns zurückschicken.

Produktname:______________________________ **Version:**_____

☐ IBM/Kompatible ☐ Apple ☐ Andere **Land:**____________

1. Wie oft benutzen Sie die Dokumentation? ☐ Häufig ☐ Gelegentlich ☐ Nie

2. Verwenden Sie bitte folgende Skala, um diese Dokumentation zu beurteilen:

	1	2	3	4	5	
Ausgezeichnet	☐	☐	☐	☐	☐	Mangelhaft
Richtig	☐	☐	☐	☐	☐	Falsch
Vollständig	☐	☐	☐	☐	☐	Unvollständig
Klar	☐	☐	☐	☐	☐	Unklar
Ansprechendes Design	☐	☐	☐	☐	☐	Nicht ansprechendes Design

3. Wurde/n das Handbuch/die Handbücher gut geschrieben, d.h. in leicht verständlicher Weise? ☐ Ja ☐ Nein
   Kommentieren Sie bitte:

____

____

4. Sind die Informationen leicht zu finden? ☐ Ja ☐ Nein

5. Der Umfang des Handbuches/der Handbücher ist: ☐ Zu groß ☐ Zu klein ☐ Angemessen

6. Vermissen Sie notwendige Informationen? ☐ Ja ☐ Nein
   Wenn "ja", welche Informationen brauchen Sie, die Sie nicht gefunden haben?

____

____

____

7. Haben Sie sachliche Fehler gefunden? ☐ Ja ☐ Nein
   Wenn "ja", geben Sie bitte den Namen des Handbuches, Seitennummer und den Fehler an:

____

____

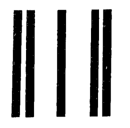
____

8. Wie beurteilen Sie folgende Teile? (Nicht alle Artikel werden Teile Ihres Produktes sein.)

	Sehr nützlich 1	2	3	4	5 ***Nicht nützlich***
Handbuch zum Lernen	☐	☐	☐	☐	☐
Handbuch zum Nachschlagen	☐	☐	☐	☐	☐
Kurzbeschreibungen	☐	☐	☐	☐	☐
Inhaltsverzeichnis	☐	☐	☐	☐	☐
Index	☐	☐	☐	☐	☐
Diagramme oder Grafiken	☐	☐	☐	☐	☐
Indexeinlagen	☐	☐	☐	☐	☐

9. Weitere Vorschläge oder Bemerkungen?

Artikelnr. 01819



Bitte freimachen

Microsoft®

16011 NE 36th Way
Dept. DRC
Box 97017
Redmond, WA
98073-9717 U.S.A.



0188 Artikelnr. 01646

Microsoft®

Microsoft® QuickBASIC 4

BASIC-Befehlsverzeichnis

0188 Artikelnr. 01646

Microsoft® QuickBASIC 4

BASIC-Befehlsverzeichnis

Microsoft®